MU6G

INSTRUCTION

SET

MANUAL

UNIVERSITY OF MANCHESTER ISSUE 2 MAY 1982

Preface: October 2019

This is a TEX copy of the MU6G Instruction Set document written by the Computer Science Department at the University of Manchester. It is the latest version that is known and is fairly complete, but it appears that some Sections were not completed, specifically Chapters 8 and 9 seem to end somewhat unexpectedly.

This version was produced by OCR conversion of a copy of the original followed by conversion into TeX format to generate this PDF. Reports of any typos or problems would be welcome. The spelling and grammar was not 'americanized', however 'data is' was left unchanged.

As mentioned in the document, MU6G was planned to be one in a family of MU6 systems, see Ch.1.1, but MU6G was the only model that was designed and brought into operation within the Computer Science Department. It was built and operated within the UM Computer Science Department from 1982 to 1987. It was then decommissioned and there is no trace of it now. The informal view was that it performed quite well and substantially met the goals described in this document.

During the development of MU6G it was decided to keep the effort strictly confidential within the UM Computer Science Department. This was due to the prior problems with the use of MU5 and other technology by ICL. ICL chose to deny any knowledge of work at Manchester. It took many years to resolve this conflict which was very costly to the University even though the final settlement was, in legal terms, in the University's favour. The cost to the wider research community was also very large and has yet to be understood. By now there is no reason that this research should be kept secret.

The software provided for MU6G was, of course, MUSS. MUSS was the system software in use on all of the computers in the UM Computer Science Department including the DEC PDP's and Motorola MC680NN-based systems and subsequently also used in UMIST. In addition, MUSS has been used under licence by a number of Companies in Europe, Canada and in the US. The legal problems with ICL made implementation of these applications more difficult as the University was still preoccupied with the ICL conflict.

However, the invention of memory paging in the Computer Science Department and its use in MUSS which demonstrated its effectiveness played a prominent role in the implementation of other paging capable processing systems, most prominently the Motorola MC680NN series and, subsequently practically all general purpose computing systems.

This document was written jointly by members of the Computer Science Department. Participants in the writing included Dai Edwards, Derrick Morris, F. H. Sumner, A. E. Knowles, J. V. Woods, C. J. Theaker, R. N. Ibbett, and S. H. Lavington. And, of course, it was inspired by Kilburn and Williams as creators of the Department which contributed a large part of modern computing.

MU6G had many innovative features, but its long development time was disadvantageous. An important example of this is the address translation using current page registers as documented in Edwards [1]. Use of associative registers and a paged segment structure as described in Theaker [2] pg.97 provides more efficient page management.

Michael D. Godfrey
michaeldgodfrey@gmail.com
Archive: archive.org

Contents

MU6G INSTRUCTION SET MANUAL i

Preface: October 2019 iii

Contents v

1 INTRODUCTION 1
- 1.1 INTRODUCTION . . . 1
  - 1.1.1 Features of the Order Code . . . 2
  - 1.1.2 Expected Performance . . . 3
- 1.2 SUMMARY OF THE ORDER CODE . . . 6

2 OPERAND ACCESSING 13
- 2.1 INTRODUCTION . . . 13
- 2.2 VARIABLE OPERANDS . . . 15
  - 2.2.1 Internal Registers Relevant to Operand Accessing . . . 15
  - 2.2.2 Accessing Variable Operands . . . 16
- 2.3 LITERAL OPERANDS . . . 17
- 2.4 REGISTER OPERANDS . . . 17

3 COMPUTATIONAL ORDERS 19
- 3.1 THE BM UNIT . . . 19
  - 3.1.1 Format for Arithmetic Data . . . 19
  - 3.1.2 Condition Bits . . . 20
  - 3.1.3 BM Orders . . . 20
- 3.2 THE A-UNIT . . . 25
  - 3.2.1 Formats for Arithmetic Data . . . 25
  - 3.2.2 Mode Bits . . . 27
  - 3.2.3 A Store and Load Orders . . . 27
  - 3.2.4 64-Bit Arithmetic Orders . . . 29
  - 3.2.5 Extended Length A Orders . . . 33

4 ORGANISATIONAL ORDERS 37
4.1 REGISTER OPERATIONS . . . 37
4.2 CONTROL TRANSFERS AND PROCEDURE CALLS . . . 38
4.3 CONDITIONAL CONTROL TRANSFERS . . . 39

5 EXAMPLES 41
5.1 ACCESSING LOCAL VARIABLES AND PARAMETERS . 41
5.2 PROCEDURE CALLING . . . 42
5.3 ACCESSING NON-LOCAL VARIABLES (AND PARAMETERS) . . . 43
5.4 ARRAYS . . . 44
5.4.1 Accessing One-Dimensional Arrays . . . 44
5.4.2 Accessing Multi-Dimensional Arrays . . . 45
5.5 FURTHER EXAMPLES . . . 46
5.5.1 Simple Loops . . . 47
5.5.2 Procedure Calling . . . 49
5.5.3 Parametric Procedures . . . 51
5.5.4 Miscellaneous Calls and Jumps . . . 53
5.5.5 Miscellaneous Examples . . . 54

6 MACHINE MODES AND INTERRUPT STRUCTURE 65
6.1 MACHINE MODES . . . 65
6.1.1 Entry to and Exit from Executive Mode . . . 65
6.1.2 Entry to Interrupt Mode . . . 66
6.1.3 Exit from Interrupt Mode . . . 67
6.2 THE INTERRUPT SYSTEM . . . 67
6.2.1 System-Based Interrupts . . . 68
6.2.2 Process-Based Interrupts . . . 70

7 VIRTUAL MEMORY MANAGEMENT 75
7.1 PAGE ADDRESS REGISTERS (PARS) . . . 75
7.2 PAR NON-EQUIVALENCE . . . 77
7.3 ADDITIONAL FEATURES OF THE MEMORY MANAGEMENT UNIT . . . 77
7.4 SUMMARY OF V-LINES ASSOCIATED WITH MEMORY MANAGEMENT. . . . 79

8 PERIPHERAL & MULTIPROCESSOR CONTROL 83
8.1 COMMUNICATION WITH PDP11 – COMPUTER INTERFACE UNIT . . . 83

8.2 PERIPHERAL INTERFACE UNITS 88
8.2.1 System Industries Disc P.I.U. 89
8.2.2 Genisco Graphical Display P.I.U. 90

9 MU6G ASSEMBLY LANGUAGE 91
9.1 A GAL PROGRAM 91
9.2 CONSTANTS 93
9.3 INSTRUCTIONS 96
9.4 OPERANDS 96
9.5 CONSTANT STATEMENTS 98
9.6 DIRECTIVE STATEMENTS 99
9.7 SPECIAL NAME 'DOT' 101
9.8 COMMENTS 101
9.9 GAL ASSEMBLER 102

Bibliography 103

Index 105

1 INTRODUCTION

1.1 INTRODUCTION

MU6G is a general purpose computer following in the tradition of MU4 (ATLAS) and MU5; however, the design objectives are different. Most previous Manchester University computers have been aimed at the 'Super-computer' market. High performance has been sought through design sophistication, which has been introduced at several levels. For example, performance has been derived from the use of

high speed arithmetic
multi-level store hierarchies
pipelines
high-level language oriented order codes.

Raw hardware speeds have been the best attainable with the available technology, but have not always quite reached those of competing machines and high performance at the user level has been the target.

In MU6 there is a change of attitude. Performance will be sought through simplicity of design in a multicomputer complex involving specialised machines (ranging from supercomputers to micros). A large general purpose MU6 system would involve several specialist machines. The 'speciality' of one machine type will be the ability to give cost effective general purpose computing up a good (but not ultra-high) level of performance (say about 1/3 CDC7600 power). This machine is MU6G; its role in the complex will be as a front end for the 'super-computers' and a user/software support system for the personal micros. More importantly for the present it will be a competitive medium performance stand alone system.

As a stand alone system MU6G will utilize a PDP11/34 as a front end, and typically will comprise

1 MU6G processor
1 megabyte of 1/2 usec store
300 megabytes of EDS storage
1 PDP11/34+Peripherals

The processor is basically a 64-bit floating point machine, but the design provides for a cheaper version with software floating point and an alternative version with 64-bit BCD arithmetic. Other variations of configuration provided for in the design include multi-processor systems and main store options from 1/2 to 4 megabytes. Initially the software will provide for simultaneous interactive use by up to 50 FORTRAN or COBOL users, but the order code also provides good support for ALGOL68-like languages, and most of the system software is written in this style of language. The user terminals may be graphics or VDU/teletypes and it is anticipated that most will be interacting with substantial software systems. Both the system software (MUSS) and the language it is written in (MUSL) have been derived from many years of research on MU5 and their good performance is proven.

1.1.1 Features of the Order Code

As in MU5 and Atlas the instructions are essentially of the form

```
F      N
```

Where F defines the function and N the operand. In MU6G there are two types of order:

1. Computational orders which operate on the accumulator or one of 3 modifier (BM) registers.

2. Organisational orders which either perform operations on one of 13 BA registers or perform control functions. In fact the BA decoding selects from 16 but 3 of these are treated specially.

The order code is designed to be uniform, simple to decode and to allow flexibility in the implementation. Accumulator, modifier and control instructions are readily distinguished and the three corresponding parts of the machine are notionally quite separate. This allows the three parts to operate autonomously in parallel to achieve good performance but does not preclude their combination in a cheaper version of the machine. It also allows flexibility in whether to provide hardware or software floating point or decimal arithmetic in a particular machine, depending the users requirements.

The order code is also designed so that the binary code generated by programs written in user languages is very compact. This, together with an equivalent compactness of the system software, is a major factor in enabling

the maximum number of on-line users to be supported with a given store size. Instructions are 32 or 16 bits long. The detailed encoding comprises a 7 bit `F`, one bit distinguishing long from short instructions and 8 or 24 operand bits. The operand forms are chosen to allow efficient implementation of all operand types frequently encountered in user languages.

The computational orders operate on 32-bit signed fixed-point items in the modifier registers, or on 64-bit floating point and/or decimal items in the accumulator register (if provided). The `F` bits define the operation to be performed, and N defines the operand.

Orders are of the single address type each combining a register with one other operand (e.g., `A` = operand, `A` + operand). Single address orders combine the advantages of simplicity both in the compilers and in the implementation of the instruction set, these both contribute to good performance. When the operand differs in size from the arithmetic size expected automatic conversion takes place, allowing short economical representations of operands where appropriate.

The organisational orders are mainly concerned with control transfers and the manipulation of organisational registers, in particular loading and storing of the `BA` registers.

An operand is specified by N, and is independent of the functions except for some restrictions on store orders and register-register moves. An operand may be a literal, a register, an operand from store or an address. Operands from store are usually accessed relative to one of the `BA` or base registers which will point to areas of store like FORTRAN labelled common, or stack frames associated with particular procedures. Control may be used as a base for accessing long literals.

1.1.2 Expected Performance

The raw hardware speed of MU6G will be two million instructions per second average. Some variation in speed will occur between different combinations of function and operand. Typical speeds are

```
fixed point add of literal                140 nsec
floating point add from cache             300 nsec
floating point multiply from cache        900 nsec
conditional branch instruction            600 nsec
subroutine entry                         1200 nsec.
```

The store cycle time is .5 micro-secs., but its performance is enhanced by a cache store which is completely invisible to the rest of the computer and the software. The above times assume a reasonable (90%) hit rate for the cache and the worst case times can be 450 nsec longer.

The overall performance will be better than a straight comparison of raw speeds would imply, because the order code structure is sympathetic to the requirements of compilers, and the general machine organisation has been decided with the needs of the operating system in mind. For example, most of the operating system paging functions are built into the hardware via a Page Address Register (P.A.R) unit which gives full cover of the store. The need for software page tables, which proved expensive in both space and time on MU5, has been eliminated. Process changing is also built-in. Even with a heavy interactive load running the operating system, demand for main store space and CPU time will be minimal. Extrapolation of MU5 figures suggest these will be 64K bytes of store (only 32K bytes resident) and less than 5% CPU time.

The biggest loss of CPU time on most interactive systems results from every activity being held up waiting for disc transfers. Clearly raw speed statistics such as channel speeds and main store size have bearing on this problem, but the logical organisation of the machine can greatly alleviate it. The features of MU6 that contribute are

small compact programs
shared code for libraries and compilers
segmented virtual store

Some feel for the capacity of the system can be gained from the store demands of typical users. The standard store of 1 Megabyte contains 512 pages. Of this the system monopolises 16 pages for its resident code and tables. If a group of users are using the commands which relate to file editing and manipulation, a further 16 pages of shared code will become locked in store due a high level of usage. Each user involved in this activity will have a virtual

machine using 5 pages. Hence the system will support $(512 - 32)/5 = 96$ users without incurring any paging traffic, except when page boundaries are encountered in the files being processed (say every 50 lines of text). The CPU demands of this class of user have been shown to be minimal. Furthermore, not all such users would be continuously active, and a logged in user simply thinking would place no demand on main store.

When users are requesting FORTRAN compilations, there is a common (shared) demand for the compiler of up to 32 pages, and the private demand of each virtual machine in compile mode will rise to 10 pages. However, it is unlikely that more than one or two users would make simultaneous compile requests. Compiling is a fairly fast activity at 300 lines/sec. It is more likely that a number of users will be simultaneously interacting with their own, or library, programs. The main contributions that the architecture makes to this situation is a reduction in total program size, by sharing the runtime package for the language, and a selection by paging only those parts of the private code and data that is currently active.

Consider the hypothetical case of a system with 50 on-line users. Two questions arise, is the CPU capacity enough to meet their demands, and can the page transfer hold-ups be kept to a tolerable level. It is best to regard the paging system as a device that will hold in main store only those pages that are actually needed. It is dangerous to regard it as a device to create an almost infinite virtual main store. Returning to the 50 users some will be inactive, that is thinking for more than 10 secs., others will be

editing
or compiling
or interacting with compiled code.

Let us assume the numbers are 16, 16, 2 and 16 respectively. The average main store available to the final category of user before paging activity begins to build up is given by

`(512 - 32 - 32 - 16 x 5 - 2 x 10)/16 =` 22 PAGES (44Kbytes).

Clearly it is not possible to say whether this is adequate without knowing what the users are doing, but many applications can be observed to run within this allocation of active pages. Considering now a possible allocation of CPU mill, if 50% were assumed to be taken up by the editing and compiling

activities each interactive user could have on average 1/3 sec. mill time every 10 seconds. It is necessary to sample the statistics of operational systems to have a feel for what this means. One such system is the DEC KA10 system at UMIST and figures produced on that system indicate that interactive users have average think times of 9 seconds and require 1.5 seconds of CPU mill time per interaction. The KA10 processor has only 1/5 the MU6G speed. Therefore we conclude that MU6G would support that type of usage with the number of users postulated. This particular comparison has been made because it is felt that the UMIST interactive users reflect the graphics/engineering design requirements that MU6G is meant to satisfy. If the heavy interactive load was eased the system could support 100 – 200 users.

1.2 SUMMARY OF THE ORDER CODE

This section summarizes the overall pattern of the order code. The detail is given in later sections as shown below.

References	Chapter
Computational Orders	Ch.3
Modifier	Ch.3.1
Accumulator	Ch.3.2
Organisational Orders	Ch.4
Operand Accessing	Ch.2
Variable Operands	Ch.2.2
Literal Operands	Ch.2.3

The seven function bits are decoded according to the following table, the first three bits selecting the column of the table, the next four the row. A brief description of each order is given below. Chapters 3 and 4 describe the orders in detail.

	0 A	1 BM1	2 BM2	3 BM3	4 CONTROL	5 BA=>	6 BA=	7 BA=REF
0	ST	STM1	STM2	STM3	JSR	RJSR	XJSR	RTS
1	STX	spare	spare	spare	VAL	STX	spare	STR
2	LDR	LDM1	LDM2	LDM3	LMS	spare	JMP	spare
3	LDD	NEQ1	NEQ2	NEQ3	spare	STA3	LDA3	LRA3
4	SP1	AND1	AND2	AND3	spare	STA4	LDA4	LRA4
5	spare	OR1	OR2	OR3	spare	STA5	LDA5	LRA5
6	LDX	OPR1	OPR2	OPR3	spare	STA6	LDA6	LRA6
7	LSR	CIN1	CIN2	CIN3	BR	STA7	LDA7	LRA7
8	ADD	ADD1	ADD2	ADD3	IEQ	STA8	LDA8	LRA8
9	SUB	SUB1	SUB2	SUB3	INE	STA9	LDA9	LRA9
10	RSB	RSB1	RSB2	RSB3	IPL	ST10	LD10	LR10
11	MLT	MLT1	MLT2	MLT3	IMI	ST11	LD11	LR11
12	DIV	DIV1	DIV2	DIV3	ILO	ST12	LD12	LR12
13	RDV	RDV1	RDV2	RDV3	IHI	ST13	LD13	LR13
14	SCL	SFT1	SFT2	SFT3	IBV	ST14	LD14	LR14
15	CMP	CMP1	CMP2	CMP3	IBC	ST15	LD15	LR15

Accumulator Orders These operate either in decimal or floating point mode.

`ST`	Store Accumulator
`STX`	Store Accumulator extension
`LDR`	Load Accumulator set to real arithmetic
`LDD`	Load Accumulator set to decimal arithmetic
`SP1`	Special functions, including standard function evaluation and type conversion
`LDX`	Load Accumulator extension
`LSR`	Load special registers
`ADD`	Add operand to accumulator
`SUB`	Subtract operand from accumulator
`RSB`	Subtract accumulator from operand, result in accumulator
`MLT`	Multiply accumulator by operand
`DIV`	Divide accumulator by operand
`RDV`	Divide operand by the accumulator, result in accumulator
`SCL`	Scale accumulator
`CMP`	Compare accumulator with operand

BM Orders These operate in integer mode. The orders for BM1 are described below, those for BM2 and BM3 are similar.

STM1	Store BM1
LDM1	Load BM1
NEQ1	BM1 not equivalenced with operand (exclusive OR)
AND1	BM1 logically ANDed with operand
OR1	BM1 logically ORed with operand

OPR1 The action of this is determined by the operand as follows
- 0 : bit select
- 2 : bit replace
- 4 : BCD pack
- 6 : BCD unpack
- 8 : BM1 = position of most significant bit of BM1

CIN1	Compare BM1 with operand and increment BM1
ADD1	Add operand to BM1
SUB1	Subtract operand from BM1
RSB1	Subtract BM1 from operand, result in BM1
MLT1	Multiply BM1 by operand
DIV	Divide BM1 by operand
RDV	Divide operand into BM1, result in BM1
SFT1	Shift BM1 by operand
CMP1	Compare BM1 with operand

Control Orders

`JSR`	Jump to subroutine
`VAL`	Validate address
`LMS`	Load machine state register (`MS`)
`BR`	Branch unconditionally (relative)
`IEQ`	Branch if equality
`INE`	Branch if not equality
`IPL`	Branch if positive
`IMI`	Branch if negative
`ILO`	Branch if negative or zero
`IHI`	Branch if positive and non-zero
`IBV`	Branch if `BM` overflow
`IBC`	Branch if `BM` carry
`RJSR`	Relative jump to subroutine
`JMP`	Unconditional jump
`RTS`	Return from subroutine

Other Organisational Orders

`STK`	Stack operand
`STA3`	Store `BA3`. Similarly `STA4` – `ST15` on `BA4` to `BA15`
`LDA3`	Load `BA3`. Similarly `LDA4` – `LD15` on `BA4` to `BA15`
`LRA3`	Load reference (i.e. address) into `BA3` Similarly `LRA4` – `LR15` on `BA4` to `BA15`
`STR`	Stack reference

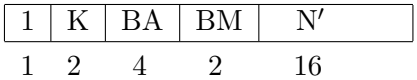
Operand Forms

There are two instruction types;

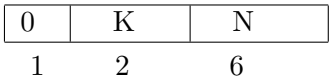
normal (32-bit), and short (16 bit).

They are distinguished by the first bit in the operand field:−

Normal



Short



In normal instructions, BA and BM are used to indicate a base address register and a modifier register, respectively (see below for decodings of BA ad BM). The meanings of K and N′ in normal instructions are as follows:−

K =	0	BA+BM+N′
	1	byte string, located at BA + 8*(BM+N′)
	2	32 bit word, located at BA + 32*(BM+N′)
	3	64 bit word, located at BA + 64*(BM+N′)

The meanings of K and N in short instructions are as follows:−

K =	0	N
	1	Register
	2	32 bit word, located at BA4 + 32*N
	3	64 bit word, located at BA4 + 64*N

Operands start on a byte boundary so the least significant 3 bits of the calculated address are ignored when accessing the operand. In the byte string operand form (K = 1, normal instruction) the number of bytes accessed (between one and eight) is specified by MS (See Ch.2.2.2).

Decoding of BM

BM =	0 use	0
	1	BM1
	2	BM2
	3	BM3

Decoding of `BA`

BA =	0 use	0
	1	Vstore (usable only in exec.mode)
	2	`CO`(address of current order)
	3	`BA3`(Stack front (`SF`)
	4	`BA4`(Namebase (`NB`)
	5	`BA5`
	6	`BA6`
	7	`BA7`
	8	`BA8`
	9	`BA9`
	10	`BA10`
	11	`BA11`
	12	`BA12`
	13	`BA13`
	14	`BA14`
	15	`BA15`

For Load-reference and Stack-reference instructions, no store access is made so the operands are interpreted as follows:

		Normal	Short
K =	0	`BA+BM+N'`	N
	1	`BA+8*(BM+N')`	Register
	2	`BA+32*(BM+N')`	`BA4+32*N`
	3	`BA+64*(BM+N')`	`BA4+64*N`

2 OPERAND ACCESSING

2.1 INTRODUCTION

Operands are divided into three classes; operands from store, literals and registers.

The powerful mechanism for accessing variable operands is appropriate for both scalars and array elements. Programming languages generally regard the store as divided into areas, for example the variables of a particular procedure, a particular FORTRAN COMMON or a particular array. The base registers `BA3` to `BA15` can be used to point to the start of such areas. Suppose `BA5` points to the variables of a procedure and a variable `I` which is the 3rd. 32-bit variable is required. Using the operand form K = 2 with `BA = 5, BM = 0` and `N = 3` the word at

```
BA5 + 32 * (0 + 3)
```

is accessed.

If `I` were to be loaded into `BM1` the assembly language language instruction would be

```
LDM1 3/5
```

(The assembly language notation used in the examples is described in Chapter 9.) The form of instruction is function mnemonic followed by operand. The operands are of the form literal `/BA/BM` where `/BM` or `/BA/BM` may be omitted if zero).

If we require to access the `I`th. element of a vector which started at word 10 we could follow this by

```
LDM2  10/5/1
```

This would access the word at

```
BA5 + 32 * (BM1 + 10)
```

Because variables of the current procedure are used very frequently a par-

ticular register `BA4` is usually set to point to these. The short operand form

```
BA4 + 32 * 3
```

enables the variable `I` to be economically accessed if it belongs to the current procedure. This short form omits the BM modification which is unnecessary for scalars.

This variable accessing method combines economy of address generation for operands in FORTRAN-like languages with the full power of recursive procedures in the Algol style of language.

Short literals of 6 or 16 bits are embedded within the instruction. Most literals occurring in programs are of this form; those which are not are typically stored with the program and accessed relative to control and the machine treats them in a way similar to variables.

Register operands are used to copy registers. They allow registers to be used in a flexible way to contain either addresses or computed results.

The operand forms are described in more detail in the remainder of this Chapter. First the transmission of operands to the appropriate unit and the form of the operand addresses is considered.

The operands for all orders are transferred from source to destination via a highway which is 64 bits wide. For a fetch order, the operand part of the order defines how the highway is loaded and the function part defines the destination (and the operation to be performed at the destination). For a store order, the function part defines how the highway is loaded, and the operand part defines the destination.

The function part of an order is described in Chapters 3 and 4; this chapter describes the operand part. With a few exceptions, which will be mentioned when they arise, any function part may be combined with any operand part, so that the two parts may conveniently be described separately.

Operands may be of various sizes up to a maximum of 64 bits. They are passed from unit to unit justified towards the least significant end of the highway, along with information about their size [32 or 64 bits]. All literals are automatically sign extended to 32 bits. Byte strings are zero filled to

the next multiple of 32 bits, thus forming a 32-bit operand given 1-4 bytes or 64-bit operand given 5-8 bytes. On a fetch order any sign extension or further manipulation of the operand is performed by the destination unit. Similarly, on a store order, unwanted bits of operand are discarded by the source unit.

Addresses are of the form:−

UNUSED	SEGMENT	PAGE	BYTE	BIT
3	8	7	11	3

Addresses in MU6G are therefore bit addresses, although in the present machine the smallest unit that can be accessed directly by the hardware is the byte. The top 3 bits are reserved for dynamic operand size information but not used in the present machine. The bottom three bits in the address of every store access, although otherwise ignored, are placed in a 3-bit register, `BITA`, which can be used by subsequent orders to aid bit selection.

2.2 VARIABLE OPERANDS

Before describing the access of variable operands in detail the action of the base and modifier registers in forming an operand address is described.

2.2.1 Internal Registers Relevant to Operand Accessing

Base address `(BA)` Registers

There are 13 `BA` registers `(BA3` to `BA15)`, each of 32 bits, used to hold a standard bit address. Two of these registers are special, from the programmers point of view. They are `BA3` and `BA4`. `BA4` is used as the base for operands in short instructions and is automatically reset by procedure entry and exit. This use of `BA4` as a base for the current procedure results in efficient implementation of recursive procedures. `BA3` is the stack front register, also used in procedure entry, which always points to the top 64-bit item on the stack (See Ch.4.2 `STK`). Register `BA3` therefore has its least significant 6 bits forced to zero. When a `BA` register is being used as an address, the bottom three bits are ignored.

The Control Register CO ('BA2')

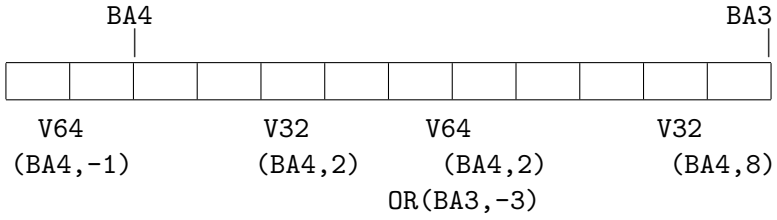
Operands may be accessed relative to CO in the same way as for BA registers except that CO is truncated to a 32 or 64 boundary for operand types K = 2 or 3 respectively. CO (BA2) therefore has its least significant 4 bits forced to zero.

The modifier (BM) registers

There are 3 BM registers (BM1 to BM3), each of 32 bits, used to calculate and hold scaled displacements from a base, as well as performing integer arithmetic and logical manipulations.

2.2.2 Accessing Variable Operands

There are three kinds of variable; they are byte string, V32 and V64, of sizes 8n (n = 1 to 8), 32, and 64 bits respectively. The operand part of the order specifies the kind of operand and also defines its name and base. The name is the distance of the variable from the base counting in units equal to the variable size. Some examples are shown below − the diagram represents a section of the virtual store marked out in 32-bit words:−



To calculate the address of the operand, the name is scaled (if necessary) and added to the base. The name displacements are always signed (and sign extended). Segment boundaries may be crossed in this calculation. This address is in fact the address of the most significant byte (bit) of the operand. Addresses within a word advance from the most significant byte or bit up to the least significant.

For this reason when it is necessary to number bits the numbering is from zero at the most significant end of a quantity.

Any variable may start on any byte boundary (although for efficiency variables should not cross a 64-bit boundary). The bottom three and top three bits of

the final 32-bit address will be ignored.

The number of bytes accessed by the byte string operand is determined by the 'byte string size' (bits 5-7 in MS). A 'byte string size' of 0 indicates a single-byte string, 1 a two-byte string and so on up to 7, which indicates an eight-byte string. This is set by the LMS instruction (See Ch.4.1) and is preserved over procedure calls. Byte string operands are zero filled on the left to 32 or 64 bits.

2.3 LITERAL OPERANDS

Literals cannot be used in store orders.

The sizes of literals are:−

6 bit
16 bit

Longer literals must be accessed by other means, such as relative to CO in a constant pool.

6 and 16 bit literals are always sign extended to 32 bits. Literals for the A unit may be further manipulated in the A unit depending upon the accumulator mode bits in AOD.

2.4 REGISTER OPERANDS

Register operands are used to read the contents of registers. They cannot be used in store orders to reset the register contents. The exception is STK, which can only be used in store orders. The particular register is encoded in the rest of the operand as:−

```
 0   %0                32  %20  A
 1   %1                33  %21  AEX
 2   %2  CO            34  %22  AOD
 3   %3  BA3   (SF)    35  %23  AEXP
 4   %4  BA4   (NB)    36
 5   %5  BA5           37
 6   %6  BA6           38
 7   %7  BA7           39
 8   %8  BA8           40  %28  EQ
 9   %9  BA9           41  %29  NE
10   %A  BA10          42  %2A  PL
11   %B  BA11          43  %2B  MI
12   %C  BA12          44  %2C  LO
13   %D  BA13          45  %2D  HI
14   %E  BA14          46  %2E  BV
15   %F  BA15          47  %2F  BC
16                     48
17  %11  BM1           49
18  %12  BM2           50
19  %13  BM3           51
20                     52  %34  LINK
21                     53  %35  BREM
22                     54  %36  BITA
23                     55  %37  MS
24                     56  %38  VALD
25                     57
26                     58
27                     59
28                     60
29                     61
30                     62  %3E  STK   {STACK V64  }
                           see also STK order (See Ch.4.2)
31                     63  %3F  UNST  {UNSTACK V64}
```

EQ, NE, PL, MI, LO, HI, and BV appear as one-bit registers, decoded from MS in the same way as in chapter4.3. BC is the one-bit B-carry register. If the condition is true a '1' is placed in the least significant bit of the operand highway, otherwise a '0' is returned. The rest of the highway is set to zeros.

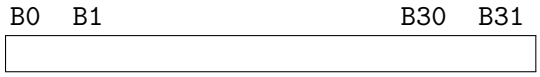
Note that although any register operand can be used with accumulator, modifier or organisational orders, some combinations may be less efficient than others (e.g. LDM1 AEX).

3 COMPUTATIONAL ORDERS

There are two types of computational orders; those which operate on the BM registers (used for modifiers to access data structures and for logical and integer arithmetic) and those which operate on the A registers (used for floating point and decimal arithmetic). There are two separate arithmetic units, the BM unit (which is also used for some of the organisational orders) and the A unit. Either or both of the floating point and decimal parts of the A unit may be omitted.

3.1 THE BM UNIT

The BM unit contains three 32-bit arithmetic registers, BM1, BM2 and BM3. The 32 bits of each register are numbered from 0 on the left hand (most significant end).

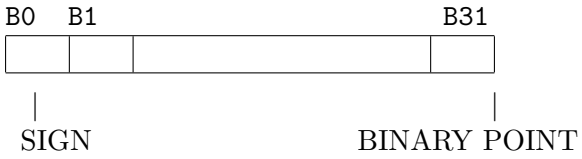


There are three other visible registers in the BM unit, namely BTOP, BC and BREM. BTOP and BREM are 32 bits long. BTOP holds the top bits of every store access longer than 32 bits, and BREM the remainder immediately after division. BC is 1 bit wide and is the carry bit from addition, subtraction and multiplication. BTOP should only be used for moving long data items around when the accumulator is not available.

3.1.1 Format for Arithmetic Data

Data is signed binary, held in 2's complement form. For multiplication and division, the binary point is at the least significant end, i.e, data is interpreted as an integer.

32-bit



RANGE $-2^{31} \leq X < 2^{31}$

On input, the top 32 bits of 64 bit items are placed in BTOP, and the least significant 32 bits used as the operand for a particular BM register. However, 32 bit items leave BTOP unchanged and only affect the BM register. On output, 64 bits are available, the top 32 being from BTOP and the bottom 32 from the BM register. If the destination is less than 64 bits wide the least significant part of the output is used and the remainder ignored.

3.1.2 Condition Bits

The condition bits are digits 1 to 4 of MS (See Ch.4.1). They are:−

1 : Overflow indicator (BV)
2 : Inhibit overflow interrupt
3 : T/=
4 : T-

If the result of any arithmetic operation is outside the range -2^{31} to $2^{31}-1$ or if division by zero takes place the overflow bit is set (and causes an interrupt unless inhibited). The overflow bit can only be cleared by the LMS order. Bits T/= and T- are set after every BM arithmetic, logical or load order. They can also be set by A compare orders and BA load orders.

T/= is 0 if and only if the result of the order is zero.
T- is 0 if and only if the result of the order is positive.

Bits 1, 3 and 4 can be tested, in various combinations, by the conditional branch commands and by some of the register operands.

3.1.3 BM Orders

Some BM orders change the condition bits T/= and T-. BM orders set BV on overflow unless otherwise stated. All BM orders may involve BTOP as specified in 3.1.1. The orders described below operate on BM1, those operating on BM2 and BM3 are similar.

STM1

The contents of the register are sent to the store at the address specified by the operand. (Does not change the condition bits).

LDM1

The operand is loaded into the BM register. (Does not change BV).

NEQ1

The register and the operand are not-equivalenced together and the result placed in the register. (Does not change BV).

AND1

The register and the operand are ANDed together and the result placed in the register. (Does not change BV).

OR1

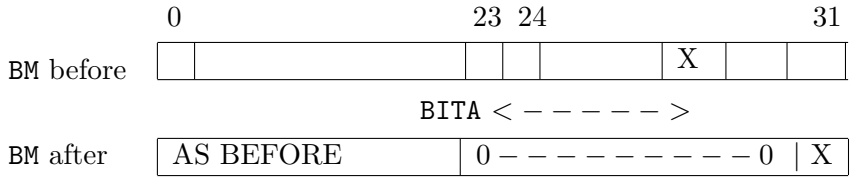
The register and the operand are ORed together and the result placed in the register. (Does not change BV.

OPR1 (BITSELECT, BITREPLACE, BCDPACK, BCDUNPACK, MOSIBIT)

The OPR1 instruction comprises the five orders listed above. The five orders are distinguished by decoding the instruction operand. The instructions themselves do not use operands except for a single bit in the BITREPLACE instruction.

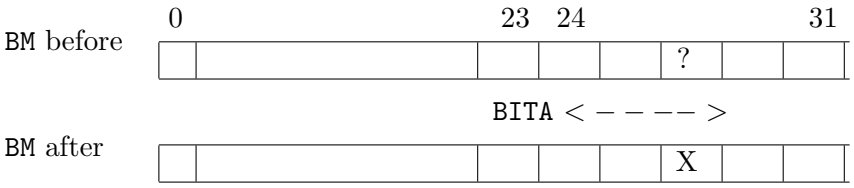
BITSELECT (operand = 0)

The bit specified by BITA is extracted from the bottom byte of the BM register, and placed in the least significant bit of the BM register. The rest of the bottom-most byte of the register is cleared. (Does not change BV).



`BITREPLACE` (operand = 2 (or 3 depending on the value to be replaced))

The bit specified by `BITA` in the bottom byte of the `BM` register is replaced by the least significant bit of the operand. The rest of the `BM` register is unchanged. (Does not change `BV`).

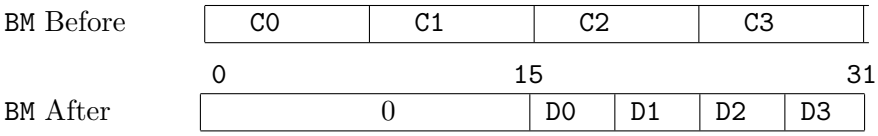


Where `X` is the least significant bit of the operand.

e.g. `OPR1` #2/0/3 (# indicates a literal operand,
in this case `BM3+2`)
where `BM3` contains either 0 or 1.
The least significant bit of `BM3` is placed in the
desired position in `BM1`.

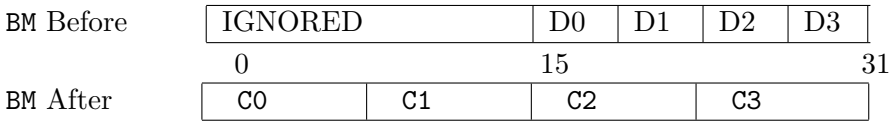
`BCDPACK` (operand=4)

The 4 bytes of the `BM` register, interpreted as decimal digit characters, are converted to BCD digits and placed in the least significant 16 bits of the register. The rest of the register is cleared. (Does not change `BV`).



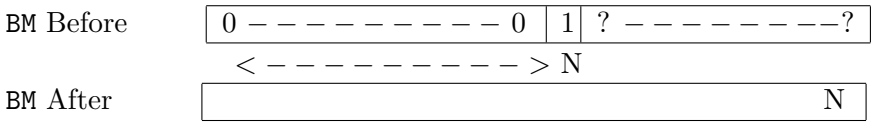
BCDUNPACK (operand=6)

The four least significant BCD digits in the `BM` register are converted to characters and placed in the register. The top four bits of each character are specified according to the expected character code, i.e. %3 for ISO. (Does not change `BV`).



MOSIBIT (operand =8)

The number of the most significant bit set in the register is placed in the register. The bits are numbered with the most significant as zero. Negative numbers are used indicate no bits set. (Does not change BV).



CMP1

The contents of the register and the operand are compared by subtracting the operand from the register contents (but not changing the register) and setting T- and T/=. An arithmetically correct result is always generated and hence BV is not changed.

CIN1

A compare operation is performed on the register, which is then incremented by one. The values of T- and T/= are determined by the compare operation. BV may be set by the increment operation.

ADD1

The operand is added to the BM register, leaving the result in the register. Any carry from the topmost bit of the result is placed in BC; that is, BC and the BM register together hold the 33-bit result.

SUB1

The operand is subtracted from the register, leaving the result in the register. BC is treated in the same way as in ADD.

`RSB1`

The register is subtracted from the operand, leaving the result in the register. BC is treated in the same way as in `ADD`.

`MLT1`

The register is multiplied by the operand, leaving the result in the register. BC is treated in the same way as in `ADD`.

`DIV1`

The register is divided by the operand, and the result, rounded towards zero, is placed in the register. (e.g. $(-8)/3 => -2$). The true remainder is placed in `BREM`. `BC` is unaffected. Note that the `BREM` may be destroyed by certain `B` operations.

The present hardware does not change the T bits on divide or reverse divide. `BV` is set by an attempt to divide by zero or by $-2^{31}/-1$.

`RDV1`

The operand is divided by the register and the result placed in the register. Otherwise, as DIV1.

`SFT1`

The register is shifted by the number of bits specified by the signed two's complement number in the bottom 6 bits of the operand (a negative operand reverses the direction of the shift). The type and amount of shift is therefore dynamically determined. The next two bibs of the operand define the type of shift:−

```
0 :  right arithmetic  ( BV unchanged)
1 :  left circular     ( BV unchanged)
2 :  right circular    ( BV unchanged)
3 :  right logical     ( BV unchanged)
```

The present hardware does not detect arithmetic overflow. If a genuine overflow is desired, MLT1 should be used. Both left and right circular shift are present because they come conveniently out of the implementation rather

than being a necessary facility.

3.2 THE A-UNIT

The A-unit can operate in floating point (R) and decimal (D) arithmetic mode. The action of arithmetic orders depends upon the mode currently selected. Only floating point arithmetic is provided on the current model of the machine.

The accumulator register is 64 bits long. There are two other visible registers in the A-unit, namely **AEX** and ADD. **AEX** is the accumulator extension register, and is 64 bits long. It serves to hold part of extended length results (See Ch.3.2.5). AOD is eight bits long, and the bits are concerned mainly with interrupts and the arithmetic mode. Also available as a register operand is **AEXP**, a copy of the exponent (bits B1-B7) of **A** in the least significant 7 bits of **AEXP**.

3.2.1 Formats for Arithmetic Data

Floating-Point (R)

Data is stored as a sign and magnitude mantissa M with a 7-bit exponent E. The most significant bit is the sign bit. It is followed by the exponent, and then the mantissa at the least significant end. The interpretation of M assumes a binary point before its most significant digit. The exponent has the base 16, and is interpreted as a 7-bit 2's complement integer plus 64 (2's complement with sign bit inverted), i.e.,

```
0000000 → -64
0000001 → -63
          .
          .
1000000 →   0
          .
          .
1111111 →  63
```

This code has been chosen so that floating-point zero has all bits = 0. The format is identical to IBM and ICL format. Floating point numbers are usually normalised and in the following all zeros is regarded as a normalized

number.

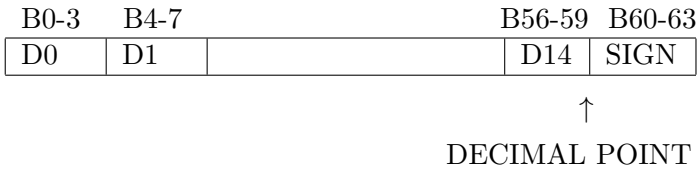
64-bit

32-BIT

This format is identical to the more significant 32 bits of the 64 bit format.

Decimal (D)

Data is stored in sign and magnitude form. The magnitude consists of 15 decimal digits occupying 4 bits each, and the sign occupies 4 bits at the less significant end. Each decimal digit is coded in binary (0 - 0000, 1 - 0001, ..., 9 - 1001), The sign code is 1101 for negative, with all other combinations meaning positive (use of 1111 is preferred). The decimal point is at the less significant end, i e., the data is interpreted as an integer.



RANGE $-10^{15} < X < 10^{15}$

3.2.2 Mode Bits

AOD is an 8-bit register. At present only the bottom 5 bits are used. They are:−

```
0-2 :  spare
3   :  Overflow indicator
4   :  Inhibit overflow interrupt
5   :  Extended length
6   :  Inhibit rounding in division
7   :  Real/Decimal (R)
```

If any arithmetic operation causes overflow, or there is division by zero, the overflow bit is set (and causes an interrupt unless inhibited). The overflow bit is only cleared by A load orders or by the LSR order.

The R bit indicates that the contents of the accumulator are in real format (R=1) or in either decimal or 2's complement integer format (R = O). It also changes the action of the arithmetic orders between real (floating point) (R = 1) and decimal (R = 0). The R bit of AOD can be set in two ways:−

a) The load orders LDR and LDD set the R bit of ADD to correspond (R or D/I)
(Note that the load order LDX assumes the current value of the R bit).

b) The order LSR may change the contents of ADD as specified by the operand.

The extended length and inhibit rounding bits are used to control multilength orders (See Ch.3.2.4 and 3.2.5).

3.2.3 A Store and Load Orders

Operands fed to the A-unit may be in any one of several formats including 32- and 64-bit real and decimal and 32-bit twos complement binary integers. (In the latter case a conversion to real form will probably follow). Results can also be in any of these forms. Only two operand sizes are permitted within the accumulator. V64 operands are treated as 64-bit objects, and all remaining operand types are 32 bits in size, except for internal register AEX. (Note that byte strings may appear to be either 32-bit or 64-bit operands, depending on their actual length : See Ch.2.1).

ST

The contents of A are sent to the store at the address specified by the operand. The data in A may be in one of three different forms: real, decimal or integer (as a result of a real to integer conversion). The destination operand may appear to the A-unit as 32 or 64 bits. If 64 bits are indicated the bits of A are sent to the store directly. If 32 bits are indicated then the the action varies with the setting of the R bit. If R is 1 (real) then the most significant 32 bits of A are placed on the least significant 32 bits of the highway and zeros on the most significant 32 bits. The least significant bit is forced to a one whenever the bits lost from the 56-bit mantissa are non-zero. When the R bit is zero, then the least significant 32 bits of the A register are sent to the least significant 32 bits of the highway, and the most significant 32 bits are set to zero. (i.e. integers are not sign extended from 32 to 64 bits).

STX

The contents of AEX are sent to the address specified by the operand. The operand size is ignored, all 64 bits being placed on the highway.

LDR

The operand is loaded into the accumulator register. If the operand size is 64 bits, then the 64 bits of the highway are loaded directly to A. If 32 bits are indicated, then the least significant 32 bits of the highway are placed in the most significant 32 bits of A, and the least significant 32 bits of A are set to zero. In both cases the operand is assumed to be real, and the order will cause the mantissa to be normalized. R is set to 1 (real). AEX is not affected by this instruction.

The overflow bit is cleared.

LDD

The operand is loaded into the accumulator register, and R set to zero. If the data item is a 32 quantity it is set to the least significant 32 bits of A, and the most significant 32 bits will be zero, i.e., the input is treated as a 32 bit decimal number. AEX is not affected by this instruction.

The overflow bit is cleared.

LDX

The 64 bits of the incoming operand are loaded directly to the AEX register. The operand size is ignored, as is R, which is unaffected by this instruction. The operand is not normalized.

The overflow bit is not cleared.

3.2.4 64-Bit Arithmetic Orders

64-bit arithmetic takes place between the data in the accumulator and that in the incoming operand, the result being placed in the accumulator. In the arithmetic orders data is assumed to be in either real or decimal format. The contents of AEX do not affect the operation, but they may be destroyed by it. In the case of real operands, the incoming operand is normalised before the order if necessary (with a consequent time penalty). All real results in the accumulator will be normalized. Since a subnormal value in A at the start of an order will be regarded as zero A should in fact always remain normalised. Rounding is performed by forcing the least significant bit of the mantissa to a one whenever the data lost from the less significant end of the mantissa is no-zero. In the case of floating point arithmetic (R = 1) and the operand size being 32-bits, arithmetic is performed to 56-bits of accuracy in the mantissa and rounded to 56 bits (and is actually slightly slower than with 64 bit operands).

The manner in which arithmetic is performed is affected by the setting of the AOD bits as described below. Any result exceeding the capacity of the result register will set bit 3 of AOD (overflow). A floating point underflow, however, will cause the accumulator to be cleared to zero. The overflow indicator will not be set.

ADD, SUB

The operand is added to/subtracted from the accumulator leaving the result in the accumulator. AEX is not affected these instructions.

The overflow bit may be set true.

RSB

The accumulator is subtracted from the operand and the result placed in

the accumulator. AEX is not affected by this instruction.

The overflow bit may be set true.

MLT

The operand is multiplied by the accumulator. If R = 1 then the more significant half of the result will be placed in A and rounded. If A underflows it will be forced to zero.

Two versions of the multiplier are possible a standard and a fast version.

The overflow bit may be set true regardless of the setting of R.

DIV

The accumulator is divided by the operand, and the quotient placed in A. If R is set and the remainder is non-zero, the least significant bit of A is forced to 1. If bit 6 of AOD (inhibit rounding) is set, the remainder is placed in AEX (regardless of R). In this case the relationships are defined by the equation

$$A = Q.OP + Rem$$

where |Rem| < |OP| and Rem is of same sign as A (as integer division in both Algol and FORTRAN). If the operand is zero, A will be unchanged.

The overflow bit may be set true, and will always be set true if OP=0.

RDV

The operand is divided by the accumulator and the quotient placed in A. If if R is set and the remainder is non-zero, the least significant bit of A is forced to 1. If bit 6 of AOD is set, the remainder is placed in AEX (regardless of R). The relationships are defined by the equation

$$OP = Q.A + Rem$$

where |Rem| < |A| and Rem is of the same sign as OP. If A is initially zero, A will be unchanged.

The overflow bit may be set true, and will always be set true if A =0.

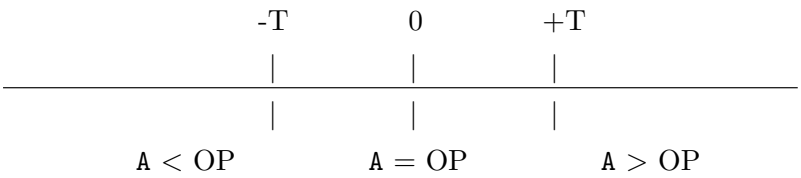
SCL

The accumulator is multiplied by the 'base' to the power of the operand, where 'base' is 2 when `R` = 1 and 10 when `R` = 0. `AEX` is not affected when `R` = 1, but may be set when `R` = 0 (`AEX` is the more significant half of a decimal number). The operand is in 2's complement form.

The overflow bit may be set true.

`CMP`

The operand is subtracted from the accumulator, a (positive) tolerance value T is subtracted from the modulus of the difference. The test bits are set from the result as illustrated.



The contents of `A` are not altered. `AEX` is destroyed.

The overflow bit will not be changed.

`LSR`

This order operates on the registers in a manner determined by the operand. Only the least significant 16 bits of the operand are significant (numbered 0 to 15 here).

If bit 8 is one then `AOD` is modified according to the equation

```
AOD 3 - 7 = AOD 3 - 7 & OP 3 - 7 OR OP 11 - 15 & NOT (OP 3 - 7)
```

i.e. for $3 \leq i \leq 7$
(a) if operand bit(i) = 1, `AOD(i)` is unaltered
(b) if operand bit(i) = 0, `AOD(i)` is set to operand bit(8 + i).

If bit 8 of the operand is a zero then bits $9 - 15$ of the operand are set into bits $1 - 7$ of `A`, i.e. `AEXP`.

SP1
FUNCTION EVALUATION

This is a set of orders operating on the accumulator, and placing the results in A. AEX will usually be destroyed. The operand determines which of the functions is performed. Additional functions can be added. The bottom 5 bits of the operand determine the function, other operand bits should be zero. In particular the top 32 bits of any 64 bit operand must be zero.

OP = 0
INTEGER TO REAL CONVERSION

A is treated as a 32 bit 2's complement integer and converted to a normalized real number in A. R is set to 1. AEX is unaltered.

OP = 1
REAL TO INTEGER CONVERSION; TRUNCATE

A is treated as a 64-bit real. If $0 <$ AEXP < 14, A is converted to a 64-bit 2's complement integer with the magnitude truncated (i.e. towards zero) in A, and a fraction in (probably unnormalized) floating point form in AEX, with an exponent of 40 (hex). R is set to zero. An integer of more than 32 bits in length will result in the overflow bit being set true. Integers greater than 57 bits cannot be formed, and cannot be distinguished from other integers that cause an overflow. If AEXP < 0, A is transferred to AEX (fractional part) and A is set to zero (integer part). If AEXP > 14, the overflow is set true, and A will not be valid.

The overflow bit may be set true.

OP = 2
REAL TO INTEGER CONVERSION; ROUND

A is treated as a 64-bit real. If AEXP < 14, A is converted to the nearest 2's complement integer. If A is exactly half way between two integers, the result will be that integer with the next highest magnitude to A. For integers of more than 32 bits in length, and for AEXP > 14, the operation is as for truncate. AEX is not defined. R is set to 0.

The overflow bit may be set true.

OP = 3
ENTIER

A is treated as a 64-bit real. If AEXP < 14, A is converted to the greatest 2's complement integer whose value is algebraically not greater than A, i.e., towards $-$ infinity. If the integer is more than 32 bits in length, or AEXP $>$ 14, the operation is as for truncate. AEX is not defined. R is set to 0.

The overflow bit may be set true.

The remaining functions with OP values in hexadecimal include:

abs(A)	%15	log(A)	%1B
sin(A)	%1E	exp(A)	%1C
cos(A)	%16	sqrt(A)	%1A
arctan(A)	%19		

The trigonometric functions are formed using a polynomial evaluation technique, and sqrt uses a Newton-Raphson iteration procedure.

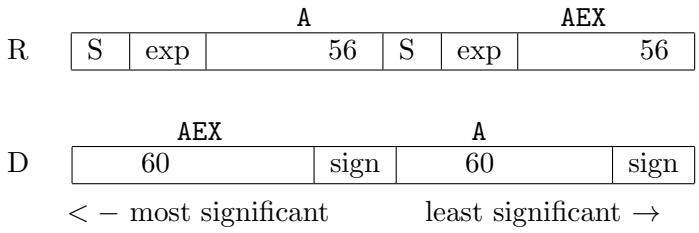
Overflow may be set true (exp(A), tan(A), (etc)).

3.2.5 Extended Length A Orders

The extended length orders are turned on by the setting of bit 5 of AOD. Only the arithmetic orders are affected by this bit, the remaining orders behaving as described previously.

FORMAT

When R is set, A and AEX form a double length number of which AEX is the less significant part. The exponent of AEX is 14 less than that of A (provided AEX is not floating point zero), and the signs are the same (i.e., the mantissa of A and AEX could be concatenated). When R $= 0$, AEX forms the more significant half of a double length number made up of AEX and A, assumed to be in decimal format. The sign digits of AEX and A are the same.



`ADD, SUB, RSB`

If `R` is 1, the operand is added to or subtracted from the number in A and `AEX` , or the number in `A` and `AEX` is subtracted from the operand. The result will be placed in `A` and `AEX` . There will no be rounding. A will be normalised, but `AEX` may not be. If `A` underflows both `A` and `AEX` are forced to zero. If `AEX` underflows but `A` does not, `AEX` will be forced to zero, (note: some valid bits may be lost in this case).

Overflow may be set true.

Note: There are no conditions attached to the relative magnitudes of `A` and the operand.

If `R` = 0 it is assumed that `A` is in decimal format. Any overflow caused by `ADD / SUB / RSB` is placed in the least significant non-sign digit of `AEX` .

Overflow cannot be changed.

`MLT`

The single length number in `A` is multiplied by the operand to give an extended length result in `A` and `AEX` . If `A` underflows both `A` and `AEX` will be forced to zero. If `AEX` underflows but `A` does not, `AEX` will be forced zero.

Overflow may be set true.

`DIV, RDV`

These operations are identical to those described for single length operations with bit 6 of `AOD` (inhibit rounding).

For `R` = 1 overflow may be set true.

For `R` = 0 overflow cannot be changed except for divide by zero which sets it true.

`SCL`

This order is identical to that described for 64-bit arithmetic, except that when `R` = 0 the overflow will not be set true, even if the result exceeds 128 bits in length.

4 ORGANISATIONAL ORDERS

The organisational orders fall into the following groups :−

(a) Register operations − orders manipulating address registers and organisational registers.
(b) Control transfers and procedure call orders.
(c) Conditional control transfers.

4.1 REGISTER OPERATIONS

Base Address Register. (BA3 to 15)

Operands and results are extended or shortened, and BTOP used, in the same way as for the BM orders (See Ch.3.2.1).

LDAn,LDn	BAn is loaded from the operand. (set T/=, T- See Ch.4.3)
LRAn,LRn	The address of the operand is placed in BAn. The top bits of BAn will be zero. (set T/=, T-)
STAn,STn	The contents of BAn are sent to the indicated address.

Machine status register MS

The machine status register contains 8 bits of system information numbered MS0 − MS7. MS0 is the executive mode bit (See Ch.6). It can only be altered by the XJSR order, or while in executive mode. MS1 − MS4 are described in chapter 3.1.2. MS5 − MS7 are described in chapter 2.2.2.

E XEC MODE	BV	INH.BV	T/=	T-	BYTE STRING SIZE
1	1	1	1	1	3

LMS This order sets various bits of MS to 0 or 1 depending upon the least significant 16 bits of the operand. If not in executive mode, MS0 is unchanged.

Number the 16 bits 0 to 15 and let $0 < i < 7$. Then
(a) if operand bit (i) = 1, MS(i) is unaltered
(b) if operand bit (i) = 0, MS(i) is set to operand bit (8 + i)

Example:

The order MS = 00110000 10100110
would set MS7 = MS1 = MS4 = 0, MS5 = MS6 = 1,
and leave MS2 and MS3 unaltered.
MS0 would be set to 1 only if the machine was in
executive mode i.e. MS0 is unaltered

MS is also altered by the RTS function (Section 4.2).

4.2 CONTROL TRANSFERS AND PROCEDURE CALLS

The link is a 64-bit register with format:−

BC	BA4	MS	BITA	CO	UNUSED
1	23	8	3	25	4

MS and BA4 are defined in Sections 4.1 and 2.2.1 respectively. CO is the 16-bit word address of the order currently being obeyed. The 23 bits of BA4 in the link give the address of a 64-bit word, i.e. the top 3 and bottom six bits of BA4 are not stored, only the 23 shown below:

BA4

3	23	6

BR Relative jump. The operand is taken to be a displacement in 16-bit units and is scaled and added to the value of CO at the start of the BR instruction.

JMP Absolute jump; CO is loaded from the operand.

JSR Absolute subroutine jump: the link is stacked, BA4 is set to BA3, and then a JMP instruction is obeyed.

XJSR Absolute subroutine jump used to enter executive. The link is stacked, BA4 is set to BA3, the operand is placed in BM1, the executive bit in MS is set, and a JMP to word 0 of segment 128 obeyed.

RJSR Relative subroutine jump: as JSR except that the final JUMP is replaced by a BR instruction, except that the value of CO used is that for the next instruction after the RJSR.

RTS This recovers the link and resets BA3 to the address of the operand.

Symbolically:-
LRA3 operand.
Set BA4, MS, BITA, and CO from the V64 link at BA4.
Bit 0 of MS is only affected if in executive mode. The top 3 and bottom 6 bits of BA4 are set to zero. (BTOP will be destroyed in the present implementation).

STK BA3 is incremented to point to the next 64-bit word in store, and the operand is written to this position. (An operand greater than 32 bits long will destroy BTOP in the present implementation).

VAL Validate the address of the operand. The permitted access is returned to a 4 bit internal register VALD. (See Chapter7).

STR BA3 is incremented to point to the next 64-bit word in store, and the address of the operand (or its value if a literal or a register) is written to this position.

4.3 CONDITIONAL CONTROL TRANSFERS

There are two test bits (T/=, T–) (See Chapters 3.1.1 and 4.1). The significance of these bits is generally as follows:−

T/= set to 0 if result = 0, 1 if result $\neq$ 0
T– set to 0 if result $\geq$ 0, 1 if result $<$ 0

A set of eight conditional jump orders is provided, six of which use the test bits, which cause a relative jump if they are suitably set. If the test succeeds then the jump is carried out in exactly the same way as for BR.

IEQ jump if T/= = 0
INE jump if T/= = 1
IPL jump if T– = 0
IMI jump if T– = 1
ILO jump if T/= = 0 or T– = 1
IHI jump if T/= = 1 and T– = 0
IBV jump if B overflow bit (BV) = 1
IBC jump if B carry bit (BC) = 1.

5 EXAMPLES

In this chapter examples of the use the instructions and operands by high-level language programs are given. Although compilers will not necessarily generate the sequences given it is intended that compiled code will be similar.

5.1 ACCESSING LOCAL VARIABLES AND PARAMETERS

Consider the Pascal-like:

```
PROCEDURE FRED (LITERAL X:REAL;  VARIABLE Y:REAL;
                LITERAL I:INTEGER;  VARIABLE C:CHARACTER;


VARIABLES       D,E,F,G : REAL;
                J,K : INTEGER;
                C1,C2 :  CHARACTER;
```

Parameters `X` and `I` cannot be used to return values, `Y` and `C` can return values.

The parameters and variables are stored thus:−
(each line represents a 64-bit word on the downward going stack)

```
        X
        V32 blank , V32 pointer to Y
        V32 blank , V32  I
        V32 blank , V32 pointer to C
BA4 → LINK
        D
        E
        F
        G
        V32 J , V32 K
BA3 → V8 C1 , V8 C2 , V48 blank
```

Within the body of the procedure

```
J = I   is achieved by   LDM1 -3/4
                         STM1 10/4

Y = D   is achieved by   LDR  @1/4
                         LD15 -5/4
                         ST   @0/15
```
(where @ indicates that a byte string operand is required)

```
C2 = C  is achieved by   LD15  -1/4
                         (LMS  #0)
                         LDM1  $0/15
                         STM1  $49/4
```
(where $ indicates that a byte string operand is required)

To generate the space on the stack for variables etc., one of the first instructions in `FRED` must be:−

```
LRA3 @6/4
```

On procedure exit, to recover the space allocated for the parameters, and to perform the return (to the address stored in the link), the last instruction in `FRED` must be:−

```
RTS @-5/4
```

5.2 PROCEDURE CALLING

If the procedure in 5.1 is called by

```
FRED (T , U , L , CH)
```
where
```
T , L and  CH are local variables and
U a VARIABLE parameter, (all of the correct type),
```

the code planted is:−

```
STK @T/4
STK U/4      [The displacement U is negative
STK L/4
STR $CH/4
RJSR →FRED
```
where `T, U` and `L` represent the displacement of these variables from `BA4`.

5.3 ACCESSING NON-LOCAL VARIABLES (AND PARAMETERS)

The simplest methods of accessing non-local variables (in languages other than FORTRAN) are via a display or via static links (See Ch.5.5.2).

```
BA4 always points to the local variables
BA3 always points to the top of stack
```

By convention, `BA5` always points to the variables of the main program. Other registers mentioned below may not be those actually used by compilers.

(Variables from intermediate levels must be accessed either via the display or via the static links).

Therefore, while in procedure `FRED` to access an integer (`V32`) variable at 9 (`V32`) positions relative to the start of the main program and place it in local variable `K`, we compile

```
LDM1 9/5
STM1 11/4
```

For FORTRAN COMMON blocks, a similar system is used. `BA6` is set to point to the blank common area and `BA7` to point to the area holding all the labelled common blocks.

e.g.
```
      SUBROUTINE  FRED
      COMMON/BLOK1/X,Y,Z
      COMMON/BLOK2/T,U,V
      .   .   .
      X=T
      .   .   .
      RETURN
      END
```

becomes FRED:

```
.  .  .
.  .  .
LDR @T+BLOK2/7
ST  @X+BLOK1/7
.  .  .
RTS @-1/4
```

5.4 ARRAYS

5.4.1 Accessing One-Dimensional Arrays

For one-dimensional arrays, there are two important ways of accessing individual elements. The first is to access elements like any other variables, as displacements from e.g. BA4.

For instance, if the origin (i.e. element zero) of an array A is to be found at BA4+n, then

A(4) is found at n+4/4

and

A(I) can be found by LDM1 I/4
n/4/1

The second method is to generate the address 'BA4+n' and store it in an extra local variable or in a base address (BA) register.

e.g. to access A(I)

```
LR15 @n/4
...
LDM1 I/4
     @0/15/1
```

or

```
LR15 @n/4
ST15 A/4
...
LD15 A/4
LDM1 I/4
     @0/15/1
```

The first method is suitable for arrays of known size (and therefore known position in store) at compile time. The second method is suitable for parametric arrays and for arrays of unknown size at compile time.

The above sequences do not perform any bound checking. If the array accesses need to be bound checked, we can check the modifier (in `BM1`) by:

```
LDM1 subscript
CMP1 lower bound
IMI  →error
CMP1 upper bound
IHI  →error
```

If the lower bound is zero or one, since loading `BM1` sets the test bits, the first `CMP` and `IMI` pair can be reduced to:−

```
IMI →error      lower bound of 0.
ILO →error        "     "    " 1.
```

5.4.2 Accessing Multi-Dimensional Arrays

The two techniques described in the previous section still apply, in conjunction with a composite subscript computed by multiplication.

E.g. `ARRAY X (1:10, 2:20, 3:30)`

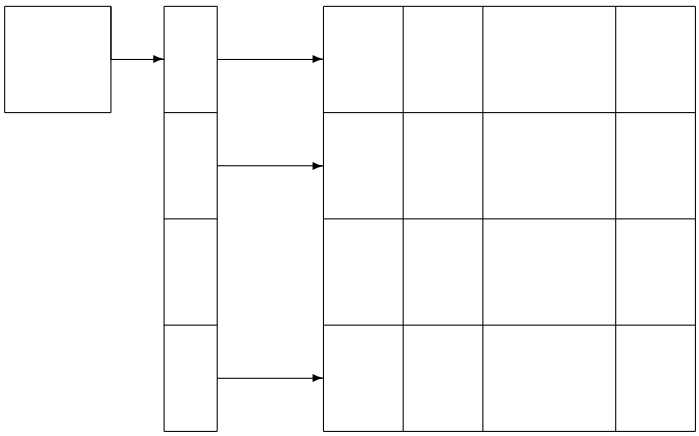
access `X(I, J, K)` by

```
LDM1 I/4
MLT1 #stride1
ADD1 J/4
MLT1 #stride2
ADD1 K/4
     @n/4/1
  or @0/15/1 as before.
```

If the bounds are constants, then the strides can be placed in the code as literals. If the bounds can be variable (e.g. parameters, or ALGOL) then the strides must be stored as variables. The set of strides (and bounds) in this case is known as a ‘dope vector’.

An alternative method is the Iliffe vector method which essentially converts a multi-dimensional access to several one-dimensional accesses. A series of auxiliary vectors containing addresses must first be generated. For a 2-D

array, a single vector is set up, with each element pointing at a separate row of the 2-D array.



For a 3-D array, each element of the 2-D array points to a strip within the 3-D array, etc.

E.g.

```
access A (I, J, K)

LD15 A/4
LDM1 I/4
LD15 0/15/1
LDM1 J/4
LD15 0/15/1
LDM1 K/4
    @0/15/1
```

5.5 FURTHER EXAMPLES

In these examples the methods in 5.1 − 5.4 will be assumed. It is assumed that registers `BA6-15, A, AEX` and the `BM` registers may be altered by procedure calls. (These registers may need to be stacked and restored if the call is in the middle of an expression).

Some trivial (single-pass) optimisations are made. For example, if few arrays are declared in a procedure, the addresses of each of them may be placed in BA registers.

('S' is used to denote a short (16-bit) order).

5.5.1 Simple Loops

a. ALGOL 60

```
        For I := 1 step 1 until N do
        A [I] := 2 * B [I] + C [I];
```

If the array addresses have been loaded into BA registers,

```
                e.g.  LD15 C/4
                      LD14 B/4
                      LD13 A/4
```

then

```
   real2:  # %4120(5)
           .  .  .
(S)        LDM1 #0
(S)        LDM2  N/4
(S)        BR→test
(S) loop:  STM1 I/4
           LDR real2
           MLT @0/14/1
           ADD @0/15/1
           ST @0/13/1
(S) test:  CIN1 BM2
(S)        IMI →loop
```

```
Otherwise
          .  .  .
(S) loop:  STM1 I/4
           LDR real2
(S)        LD14 B/4
           MLT @0/14/1
(S)        LD15 C/4
           ADD @0/15/1
```

```
(S)        LD13 A/4
           ST @0/13/1
   test:    .  .  .
```

FORTRAN (Note that in FORTRAN the above optimisation is only necessary for COMMON blocks and for parameters).

```
DO 1  I = 1, N, 1
A(I ) = 2 * B(I) + C(I)
1 CONTINUE
```

becomes

```
(S)        LDM1 #0
(S)        LDM2 N/4
(S)        BR→test
(S) loop:  STM1 I/4
           LDR real2
           MLT @B/4/1
           ADD @C/4/1
           ST  @A/4/1
(S) test:  CIN1 BM2
(S)        IMI→loop
```

b. ALGOL 60

```
For I := 2 step 1 until 99 do
C[I] := (E[I-1]+E[I+1]) * K1+E[I]*K2;
```

```
(S)        LD15 C/4
(S)        LD14 E/4
(S)        . .  .
(S)        LDM1 #2
(S) loop:  STM1 I/4
           LDR @-1/14/1
           ADD @1/14/1
(S)        MLT @K1/4
(S)        ST  STK
           LDR @0/14/1
(S)        MLT @K2/4
```

```
     (S)        ADD UNST
                ST @0/15/1
                CIN1 #99
     (S)        IMI→loop
```

FORTRAN

```
       DO 1 I = 2, 99
     1 C(I) = (E(I-1)+E(I+1)) * K1 + E(I) * K2

     (S)        LDM1 #2
     (S) loop:  STM1 I/4
                LDR @E-1/4/1
                ADD @E+1/4/1
     (S)        MLT @K1/4
     (S)        ST STK
                LDR @E/4/1
     (S)        MLT @K2/4
     (S)        ADD UNST
                ST @C/4/1
                CIN1 #99
     (S)        IMI →loop
```

5.5.2 Procedure Calling

Displays

A register is set pointing to the variables of each textual level.

FORTRAN, Innermost ALGOL procedures (using a display).

```
enter by      STK parameter 1
               .
               .
               .
              STK parameter n
              RJSR →procedure name
   name :    LRA3 @ number of variables /4
               .
               .
               .  body of procedure
```

```
          .
        RTS @- (number of parameters + 1)/4   [= @-(n+1)/4
```

Non-innermost ALGOL procedures (using a display)

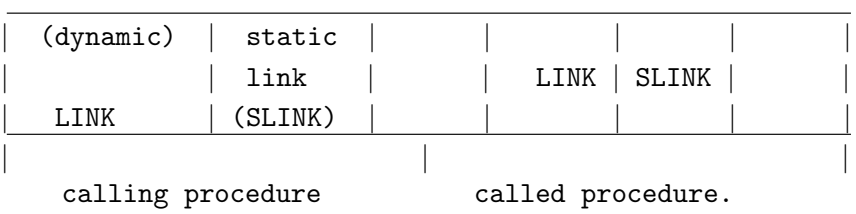
enter as above

```
name :  LRA3 @ number of variables/4
        STAx →STK
        LDAx BA4   [x in range 5 to 15 = textual level+5
           .
           .
           .  body of procedure
           .
        LDAx = UNST
        RTS @- (n+1)/4
```

Chained static links

Each procedure remembers a static link pointing to the variables of the procedure, with lower textual level, that contains its declaration.

ALGOL procedures (using chained static links)



A slight variation on a full static linking is used; the variables of the main program (textual level zero) are always pointed at by a specific BA register (BA5 is used by convention). Therefore static links pointing back to the main program are not necessary and are never generated or saved. Apart from this one case (i.e. calling a procedure at textual level one) the calling code is as follows:−

The parameters are stacked as normal.

The static link (if needed) is generated in a BA register (BA12 is used by

convention). This is obtained as follows:−

Case new textual level – current textual level of

```
 +1 :  LD12 BA4
  0 :  LD12 SLINK/4
 -1 :  LD12 SLINK/4
       LD12 SLINK/12
 -2 :  etc.

RJSR →procedure

procedure :      ST12 STK
                 LRA3 @number of variables/4
                 .
                 .  body of procedure
                 .
                 RTS @ -n -1/4
```

5.5.3 Parametric Procedures

ALGOL procedures (with chained static links)

The parameter for a parametric procedure consists of the `V32` address of the procedure and the `V32 SLINK` that needs to be planted. (The `SLINK` is generated in the same way as the above example). The parametric procedure is called by:−

```
STK  parameters as before
LD12 SLINK part of procedure parameter
JSR  address part    "    "     "
```

ALGOL procedures (with display)

Because of a pathological case (a jump to a parametric label through a parametric procedure followed by a non-local access) this case is difficult to explain. An example will be given.

```
begin
 ...
procedure p1 (x);
real x;
value x;
begin
 ...
end;
 ...
procedure p2 (p);
procedure p;
begin
...
p(z);
...
p2 (p1);
...
end;
```

becomes

```
    ...
p1 :  LRA3 @ number of variables/4
    ...
    RTS @-2/4
    ...
p2 :  LRA3 @ number of variables/4
    ...
    STK @ x/4 [P's parameter
    STK -1/4  [display vector address
    JSR -2/4  [p's address
    ...
    RTS @-2/4
    ...
    LR15 Display vector/4
    ST15 STK
    LR15 pp1
    ST15 0/3
    Copy display held in BA-registers to Display vector
    RJSR →p2
    BR →Label
```

```
pp1 :  stack display held in BA-registers
    fill     "     " " "     " from Display vector
                               [address in -1/4
    STK @ -2/4   [i.e.  the parameter for p1
    RJSR →p1
    unstack display
    RTS @-3/4
Label:  ...
```

In effect, an extra procedure (pp1) is used purely to ensure that the display is correct.

5.5.4 Miscellaneous Calls and Jumps

Relocatable code

```
GOTO L becomes BR →L

GOTO (L1,...,Ln), I becomes
               BR  I/4
               BR →L1
                    .
                    .
                    .
               BR →Ln
```

It is not presently proposed to generate relocatable switches from MUTL.

```
CALL P becomes  RJSR → P

CALL organisational proc  P becomes  XJSR jump table item

CALL library proc P becomes  JSR jump table item
```

Jump table

```
        JSR jump table item
       ...
item :   JMP (or  BR →) proc  [part of jump table]
       ...
proc : as normal
       ...
        RTS
```

Non-local jump with static links

These use label variables which contain an address and the value of stack front (BA3). At the jump, BA3 is reset. At the label, BA4 is reset from BA3.

5.5.5 Miscellaneous Examples

FORTRAN BUBBLE SORT

```
      SUBROUTINE sort (n,a)
      DOUBLE a(n), temp
      INTEGER n, i, j
      DO 10 i = 1, n-1
         DO 10 j = i+1, n
           IF (a(i).LE.a(j)) GOTO 10
              temp = a(i)
              a(i) = a(j)
              a(j) = temp
 10   CONTINUE
      RETURN
      END

(S)        LR3 @3/4
(S)        LD13 N/4
(S)        LD15 A/4
           LDM3 0/13
(S)        LDM1 #1
(S) L1:    STM1 I/4
           LDM2 #1/0/1
```

```
(S) L2:    STM2 J/4
           LDR @0/15/1
           CMP @0/15/2
(S)        ILO →L10
(S)        ST  @TEMP/4
           LDR @0/15/2
           ST  @0/15/1
(S)        LDR @TEMP/4
           ST  @0/15/2
(S) L10:   CIN2 BM3
(S)        IMI →L2
           CIN1 #-1/0/BM3
(S)        IMI →L1
(S)        RTS @-3/4
```

Accessing a packed array of bits

X a packed array of bits

```
X[J] := X[I]
(S)           LDM1 I/4
              LD15 #X/4/1
(S)           (LMS #0)
              LDM2 $0/15/0
(S)           OPR2 #Bitselect
(S)           LDM1 J/4
              LD15 #X/4/1
              LDM3 $0/15/0
              OPR3 #Bitreplace/0/BM2
              STM3 $0/15/0
```

ACCESSING FIELDS

X an array (from 0 to 100) of 19-bit items, starting on any arbitrary bit boundary

```
X(J):=X[I]+K;
```

```
LMS #%F803 i.e.  Size = 4 bytes
LDM1 I/4
MLT1 #19
LD15 #X/0/1

LDM2 $0/15
LDM1 BITA
SFT2 #(LEFT.CIRCULAR+19)/0/1
AND2 #%7F(4)

ADD2 K/4

LDM1 J/4
MLT1 #19
LD15 #X/0/1

LDM3 $0/15
AND2 #%7F(4)
LDM1 BITA
SFT3 #(LEFT.CIRCULAR+19)/0/1
AND3 #%F(3)80(4)
OR2 BM3
SFT2 #(RIGHT.CIRCULAR+19)/0/1
STM2 $0/15
```

PASCAL pointers and data structures.

```
Type Activeunit = Record
                    wantip, wantop, outbuff :  integer:
                    abuffer :  array [ ] of ^ bufferstatus;
                    suc :  array [ ] of ^ activeunit;
                  end
Function findop (unit :  ^ activeunit) :  boolean:
Var J : integer:
     Found :  boolean:
begin
  with unit ^ do
    begin
      if wantop in [0..  nbuff] then
        begin
```

```
          J := wantop
          with suc[J] ^ do
          found := not ((wantip in [0..  nbuff]) and (wantip <> J)
                      or (abuffer[J] <> nil));
        end
      else
          begin
            J := -1; found := false;
            repeat
               J := J+1;
               if suc[J] <> nil then
                  with suc[J] do
                     found := <expression>;
            until found or (J = nbuff)
          end:
      if found then
        outbuff := J;
      end:
   findop := found:
end:
```

```
(S)          LRA3 @2/4
(S) with     LD14 UNIT/4
    if       LDM1 WANTOP/14
(S)          IMI →else
             CMP1 NBUFF/5 [Global variable
(S)          IHI →else
(S) then     STM1 J/4
    with     LD15 SUC/14/1
    in       LDM2 WANTIP/15
(S)          LDM3 PL
             CMP2 NBUFF/5
(S)          AND3 LO
    and      CMP1 WANTIP/15
(S)          AND3 NE
    or       LDM2 ABUFFER/15/1
(S)          OR3 NE
(S) not      NEQ3 #1
(S)          STM3 FOUND/4
(S)          BR →end2
```

```
(S) else    LDM1 #-1
(S)         STM1 J/4
(S)         LDM2 #0
(S)         STM2 FOUND/4
(S) repeat  ADD1 #1
(S)         STM1 J/4
    if      LD15 SUC/14/1
(S)         IEQ →fi
            (LDM2 <expr>)
(S)         STM2 FOUND/4
(S) fi      CMP2 #0
(S)         INE →end1
            CMP1 NBUFF/5
(S)         INE →repeat
(S) end1    LDM2 FOUND/4
(S)         IEQ →end2
            STM1 OUTBUFF/14
(S) end2    LDM1 FOUND/4
(S)         RTS @-2/4
```

```
if Y in X then ...  else ...; [Y and X pascal sets.]

(S)         LDM1 X/4
(S)         NEQ1 #-1
(S)         AND1 Y/4
(S)         INE →else
   then:...
   else:...
```

Extended-length Real Arithmetic

These procedures are an example of the way multi-length arithmetic can be implemented. They are based on the ATLAS double-length arithmetic extracodes. -T and -B are used to indicate the top (most significant) and bottom 64 bits of variables, respectively. The `LSR` instructions act on `AOD`.

```
Load acc (XT, XB)     load X into accumulator (A, AEX)
```

```
LDR XB/4
LSR #Extended length
LDX #0
ADD XT/4
RTS @-3/4

Store acc (V32 address of XT) store accumulator in X

LD15 address of XT/4
ST  @0/15
STX @1/15
RTS @-2/4

Negate acc ( )

LSR #Normal length, rounding
ST  STK
LDR AEX
RSB #0
LSR #Extended length
LDX #0
SUB UNST
RTS @-1/4

Add acc (XT, XB)     add X into accumulator

(V64 W, Y, Z)
LR3 @3/4
LSR #Normal length, rounding
ST  @Z/4
LDR AEX
ADD @XB/4 (*)
ST  @Y/4
LDR @Z/4
LSR #Extended length
LDX #0
ADD @XT/4 (*)
ST  @W/4
LDR AEX
LDX #0
```

```
ADD @Y/4
ADD @W/4
RTS @-3/4

Subtract acc (XT, XB)    subtract X from accumulator

As Add acc except that instructions marked with (*) become
SUB @XB/4,  SUB @XT/4 respectively.

Mult acc (XT, XB)    multiply accumulator by X.

(V64 U, W, Y, Z)
LRA3 @4/4
LSR  #normal length, rounding
ST   @U/4
LDR  AEX
MLT  @XT/4
ST   @W/4
LDR  @U/4
MLT  @XB/4
ADD  @W/4
ST   @Y/4
LDR  @XT/4
LSR  #Extended length
MLT  @U/4
ST   @Z/4
LDR  AEX
ADD  @Y/4
ADD  @Z/4
RTS  @-3/4
```

COBOL

1. Simple move with equal length operands.

```
     A PIC X(5)
     B PIC X(5)

     MOVE A TO B
```

```
LMS #4
LDD $A/BA4
ST  $B/BA4
```

2. Simple move with unequal length operands.

```
C PIC X(7)
D PIC X(20)

MOVE C TO D

LMS #6
LDD $C/BA4
ST  $D/BA4
LDD #0
LMS #7
ST  $C+7/BA4
LMS #4
ST  $C+15/BA4
```

3. Table accessing.

```
          STUDENT-RESULTS OCCURS 3 TIMES
                  INDEXED BY YEAR
          COURSE OCCURS 6 TIMES
               COURSE-TITLE PIC X(6)
               COURSE-MARK  PIC 99 COMP-3 SYNC
          AVERAGE PIC 99V9 COMP-3 SYNC
     COURSE-TOTAL PIC 9(3) COMP-3 SYNC RIGHT
     COURSE-NO    PIC 9    COMP   SYNC RIGHT

  MOVE ZERO TO COURSE-TOTAL
  PERFORM ADD-MARK-TO-COURSE-TOTAL
          VARYING COURSE-NO FROM 1 BY 1
          UNTIL COURSE-NO IS EQUAL TO 7
  DIVIDE COURSE-TOTAL BY 6 GIVING AVERAGE (YEAR) ROUNDED
ADD-MARK-TO-COURSE-TOTAL:
  ADD COURSE-MARK (YEAR,COURSE-NO) TO COURSE-TOTAL
NEXT-PARA:
```

```
                LDD  #0
                ST   COURSE-TOTAL/4

                LR15 RETURN
                ST15 LABEL/4
                LDM2 #1
LOOP:           STM2 COURSE-NO/4
                CMP2 #7
                IEQ  →RESET-RETURN
                BR   →ADD-MARK-TO-COURSE-TOTAL
RETURN:         LDM2 COURSE-NO/4
                ADD2 #1
                BR   →LOOP
RESET-RETURN:   LR15 NEXT-PARA
                ST15 LABEL/4

                LDD  COURSE-TOTAL/4
                SCL  #2
                DIV  #DECIMAL-6
                ADD  #DECIMAL-5
                SCL  #-1
                LDMI YEAR/4
                ST   AVERAGE/4/1

ADD-MARK-TO-COURSE TOTAL:
                LDM1 COURSE-NO/4
                MLT1 #STRIDE
                ADD1 YEAR/4
                LDD  COURSE-MARK/4/1
                ADD  COURSE-TOTAL/4
                ST   COURSE-TOTAL/4
                JMP  LABEL/4
```

4. Conversion.

```
                E PIC 999
                F PIC 999 COMP-3

                MOVE E TO F
```

```
                LMS   #2
                LDM1  $E/4
                OPR1  #BCDPACK
                LMS   #1
                STM1  $F/4
```

5. Single-character scanning.

```
                G PIC X (12)
```

INSPECT G REPLACING ALL LEADING SPACES BY ZEROES

```
                LDM1  #0
                LDM2  #'0
                LDM3  #'
                LMS   #0
LOOP:           CMP3  $G/4/1
                INE   →OUT
                STM2  $G/4/1
                CIN1  #11
                IMI   →LOOP
OUT:
```

6. Multiple-character scanning

```
                H PIC X(35)
```

COUNT PIC 9(2) COMP SYNCRIGHT

INSPECT H TALLYING COUNT FOR CHARACTER 'XYZ'

```
                LDM1  #0
                LMS   #2
                LDM2  #'XYZ
LOOP:           CMP2  $H/4/1
                CIN1  #33
                IMI   →LOOP
                LDM1  #35
OUT:            STM1  COUNT/4
```

6 MACHINE MODES AND INTERRUPT STRUCTURE

6.1 MACHINE MODES

The machine has three execution modes, which govern at any given time:

(a) The access rights given to the processor by the memory management system.

(b) The response of the processor to interrupts.

The mode is controlled by the setting of the EXECUTIVE bit in the machine status register, and the INTERRUPT bit in the central machine control V-line, V.MC.CONTROL. The three modes are called user mode (EXECUTIVE = INTERRUPT = 0), executive mode (EXECUTIVE = 1, INTERRUPT = 0) and interrupt mode (EXECUTIVE = INTERRUPT = 1).

The effect of the mode on the memory management and interrupt hardware will be described in detail later. In brief, the EXECUTIVE digit enables a process to access parts of the store (including the V-store) which are not normally accessible in user mode, while the INTERRUPT digit has the effect of inhibiting interrupts. The EXECUTIVE bit is always set when in interrupt mode.

6.1.1 Entry to and Exit from Executive Mode

The XJSR instruction results in a transfer of control to the operating system, in executive mode. This is the means whereby processes executing in user mode may request services from the operating system. The mechanism is secure in that (a) the entry point is fixed, and hence only valid operating system procedures may be entered in this way; (b) the XJSR instruction records in the link whether the call was from user or executive mode, thus enabling the RTS operation to restore the mode correctly; (c) an RTS instruction can never give rise to an increase in privilege.

Software forced interrupts (see later) also result in an entry to executive mode.

6.1.2 Entry to Interrupt Mode

Entry to interrupt mode normally takes place as a result of an interrupt. Whenever interrupt mode is entered, the machine state (including control, the machine registers, and the status) is dumped into a register dump associated with the current process, and some of the registers (control, status, NB, VM.TIMER) are re-loaded with pre-defined values for servicing the interrupt. The real address of the register dump is given by the V-line V.PRB.A (Process Register Block Address). The registers are dumped on interrupt entry in the following format:

	V32		
V.PRB.A 0	BA3	(%1B) 27	
1	BA5	(%1C) 28	
2	BA6	(%1D) 29	
3	BA7	(%1E) 30	V.VM.INTS
4	BA8	(%1F) 31	V.VM.TIMER
5	BA9	(%20) 32	LINK } (BA4/MS)
6	BA10	(%21) 33	LINK } (CO)
7	BA11	(%22) 34	ACC }
8	BA12	(%23) 35	ACC }
9	BA13	(%24) 36	AEX }
(%A) 10	BA14	(%25) 37	AEX }
(%B) 11	BA15	38	
(%C) 12	BM1	39	
(%D) 13	BM2	40	
(%E) 14	BTOP }	41	
(%F) 15	BM3 }	42	
(%10) 16	VALD	43	
(%11) 17	AOD	44	
(%12) 18	BREM	45	
(%13) 19			
(%14) 20			

On interrupt entry, after performing a register dump to [V.PRB.A], the reason for interrupt is loaded into BA15 before restoring the operating

system registers NB, SF and LINK, from the hardware dump. The values stored in BA15 are as follows:

BA15 value	Interrupt value
%8	SYSTEM ERROR
%9	PROGRAM FAULT
%A	VOLUNTARY INTERRUPT
%B	VIRTUAL STORE INTERRUPT
%C	VALIDATE FAIL INTERRUPT
%D	VIRTUAL TIMER INTERRUPT
%E	SOFTWARE FORCED INTERRUPT
%F	SYSTEM BASED INTERRUPT

Since virtual store interrupts are not allowed in interrupt mode, all interrupt mode code uses a fixed core-resident data area. The register re-loading operation described above automatically switches the processor to using this area as its stack. In some circumstances it is necessary for the operating system to make a deliberate switch into interrupt mode. This is achieved by writing a '1' to the interrupt digit in V.MC.CONTROL. The effect is exactly the same as with involuntary interrupt entries.

6.1.3 Exit from Interrupt Mode

Exit from interrupt mode back into a process is achieved by writing '0' to the interrupt digit in V.MC.CONTROL. The values of NB, SF and a re-entry link for the operating system are saved in special registers and then the process' state is completely restored from the values dumped at [V.PRB.A].

6.2 THE INTERRUPT SYSTEM

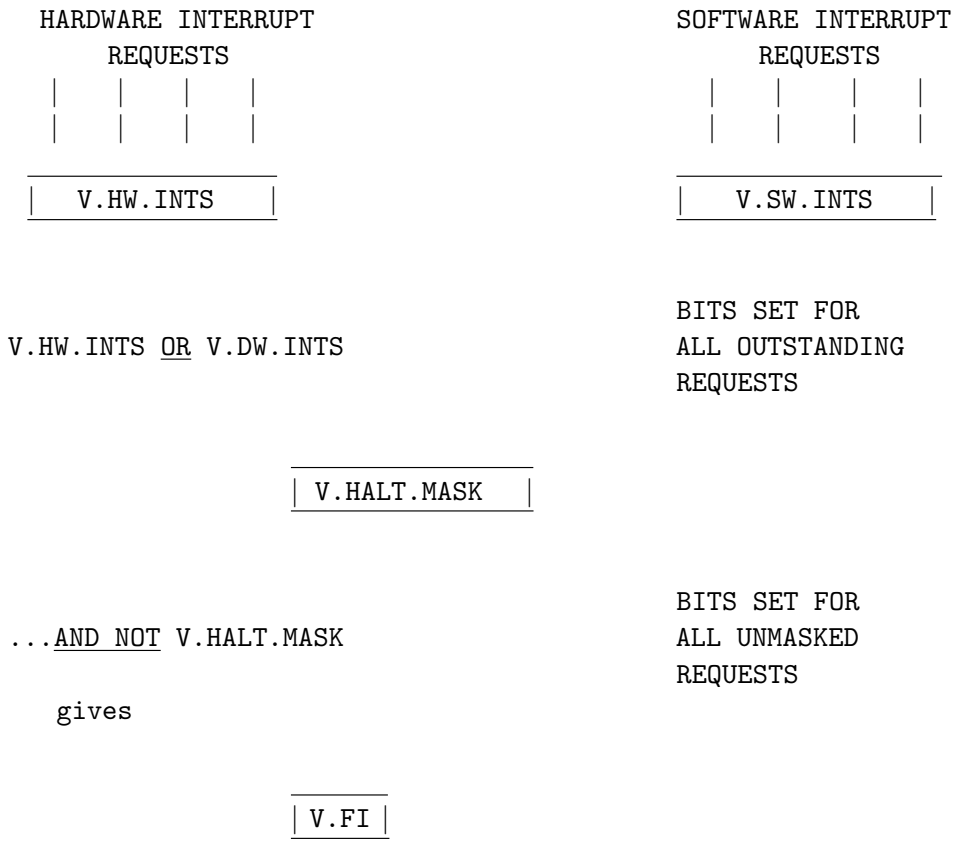
Interrupts are classified into two groups, according to their source. Process-based interrupts are those whose occurrence is directly attributable to some action of the currently executing process. System-based interrupts are those which result from asynchronous events occurring outside the CPU. The usual response to an interrupt is a switch to interrupt mode, with the associated dumping of machine state. As interrupts are not usually permitted while the machine is in interrupt mode, the response of the system to each type of interrupt is summarised in the following table:

	CURRENT MODE		
	USER	EXECUTIVE	INTERRUPT
INTERRUPT TYPE / PROCESS BASED	FORCE INTERRUPT	FORCE INTERRUPT	SYSTEM ERROR
SYSTEM BASED	FORCE INTERRUPT	FORCE INTERRUPT	INTERRUPT REMAINS PENDING

INTERRUPT ACTIONS

6.2.1 System-Based Interrupts

System interrupts generally arise as a result of asynchronous events occurring outside the CPU, such as the completion of a peripheral transfer or an interrupt from the real-time clock. Each potential source of interrupts has the ability to set an interrupt request digit in the CPU. This ultimately results in the scheduling of a process to deal with the request. This is controlled by a set of V-lines in the CPU, illustrated in the following figure.



SYSTEM INTERRUPT STRUCTURE

The V-lines are:

	V.HW.INTS	(16 bits, read-only)
	V.SW.INTS	(32 bits, read/write)
	V.HALT.MASK	(32 bits, read/write)
	V.FI	(32 bits, read-only)

`V.HW.INTS` corresponds to the row of hardware interrupt request digits, where each bit is associated with a group of similar devises. The digits are ordered according to priority, with the highest priority requests at the most significant end. This V-line may only be read by the `CPU`, as bits are extinguished by

writing to the V-store of the interrupting device.

`V.SW.INTS` is a read/write V-line, whose most most significant 16 digits correspond exactly to those in `V.HW.INTS`. By writing to this V-line, the software is able to create the effect of an interrupt. The least significant 16 bits correspond to user processes and have no equivalent in the hardware interrupt register.

The logical 'OR' of these two lines gives all the currently outstanding interrupt requests.

V.HALT.MASK is used to control which of the above requests are to be made available for scheduling. A '1' in digit i of V.HALT.MASK inhibits the appearance of the corresponding digit in the next register, V.FI. This allows a process servicing interrupts to halt itself without danger of being freed by a subsequent interrupt. For example, a process reading input might need to halt awaiting completion of a transfer of data to the backing store.

V.FI, the Free Indicator, forms the basis of all low-level CPU scheduling in the system. The Free Indicator is formed as:

```
V.FI = (V.HW.INTS OR V.SW.INTS) AND NOT V.HALT.MASK.
```

Each non-zero digit in `V.FI` thus represents an outstanding request which is free to be scheduled.

If any of the most significant 16 bits in V.FI is non-zero, and the machine is not currently executing in interrupt mode, a system-based interrupt is forced.

6.2.2 Process-Based Interrupts

There are six process-based interrupts, each with its own distinct servicing procedure:

(1) Virtual Store Interrupt
(2) Validate Fail Interrupt
(3) Program Fault Interrupt
(4) Virtual Machine Timer Interrupt
(5) Voluntary Interrupt Entry
(6) Software-Forced Interrupt.

The table below indicates the modes in which each type of interrupt may arise, and the resulting CPU action.

		CURRENT MODE		
		USER	EXEC	INT
INTERRUPT TYPES	1,2,3,5	ENTER INT MODE PROCEDURE	ENTER INT MODE PROCEDURE	SYSTEM ERROR
	4	ENTER INT MODE PROCEDURE	INTERRUPT REMAINS PENDING	IGNORE
	6	ENTER EXEC MODE PROCEDURE	INTERRUPT REMAINS PENDING	INTERRUPT REMAINS PENDING

PROCESS INTERRUPT RESPONSES

1) Virtual Store Interrupt

   The virtual store interrupt occurs on any exception condition in the virtual addressing (i.e. not in core, or illegal access attempted). The V-store and actions associated with virtual store management are described in the following chapter.

2) Validate Fail Interrupt

   This interrupt occurs when a validate (`VAL`) order is unable to check access (i.e. page not in core).

3) Program Fault Interrupt.

   This signals an error condition (e.g. arithmetic overflow, etc.) in the currently-executing process. The type of error is indicated in a bit-significant V-line, `V.PF.STATUS`.

types of program fault:-
Floating Point Error (overflow)
Fixed Point Error (overflow)
Illegal order

4) Virtual Machine Timer Interrupt

A read/write V-line `V.VM.TIMER` is used to measure the passage of 'virtual time' in a process. It is decremented after every instruction, and an interrupt occurs whenever it is non-positive in user mode. `V.VM.TIMER` is preserved and restored along with the other registers on transitions in and out of interrupt mode. On entry to interrupt mode, it is set to its maximum value and continues to count, but will not interrupt.

5) Voluntary Interrupt Entry

This process-based interrupt occurs as a result of explicitly writing '1' to the interrupt digit in `V.MC.CONTROL`.

6) Software Forced Interrupt

The software forced interrupt occurs whenever the V-line `V.VM.INTS` is non-zero and the machine is executing in user mode. The register `V.VM.INTS` is preserved in the register dump on entering interrupt mode. It consists of up to eight bits, which are set by the software, to generate interrupt conditions within the user's virtual machine. The effect of a software forced interrupt is similar to that of a forced `XJSR` call to a system procedure in executive mode.

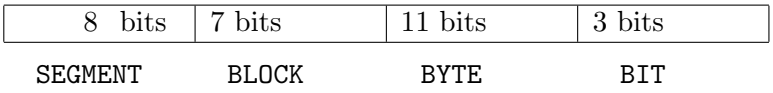
Summary of V-lines associated with interrupts.

Name	64bit word V-address	write/read	format
`V.SW.INTS` Software interrupt request	%0000	W/R	32 individual bits 1 = outstanding interrupt
`V.HALT.MASK`	%0001	W/R	32 individual bits 1 inhibits interrupt appearing
`V.FI` Free Indicator	%0002	R	32 individual bits 1 = outstanding non-inhibited interrupt
`V.VM.TIMER` Instruction timer	%0003	W/R	16 bits Interrupts when counted to zero. Counts 1 for every CO update
`V.PRB.A` PRB for current process	%0004	W/R	32 bits as bit address only bits 03 to 20 used others forced = zero
`V.VM.INTS` Software forced interrupts	%0005	W/R	8 individual bits
`V.PF.STATUS`	%0006	R	2 bits. l.s = B-overflow m.s = Acc overflow zero = ILLEGAL INSTRUCTION
`V.MC.CONTROL`	%0007	W/R	1 bit

Name	64bit word V-address	write/read	format
`V.HW.INTS` Hardware interrupt requests Allocation 34 - 41 systems bus interrupts	%0008	R	16 individual bits (Most significant of 32) positions 0-7 are V-store units 6 - D
`V.DCU.MESS`	%000C	W/R	8 bits Message buffer to/from `DCU`
`V.DCU.OBSY`	%000D	W/R	1 bit Output to `DCU` busy
`V.DCU.IBSY`	%000E	W/R	1 bit Input from `DCU` busy.
`V.SB.INITS`	%F000	R	8 individual bits, one per system bus position. Bit 56 Position 0 Bit 59 Position 3 P.I.U/S.I Disc Bit 61 Position 5. C.I.U/PDP 11 Bit 63 Position 7.

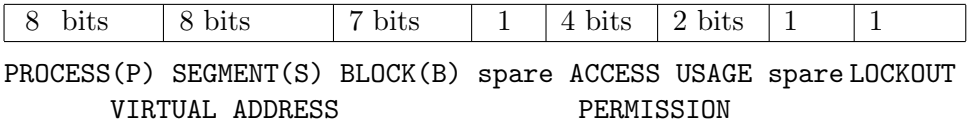
7 VIRTUAL MEMORY MANAGEMENT

The memory management system of the MU6G machine provides a paged segmented virtual store for each process. The virtual store consists of 256 segments, each of 256 Kbytes, of which the segments 128-255 are common to all processes. Each segment is further subdivided into pages of 2 Kbytes. Addresses in the machine are bit addresses, although in the present proposal, the smallest addressable unit is the byte. A virtual address therefore has the form:



7.1 PAGE ADDRESS REGISTERS (PARS)

The translation from a virtual to a real address is performed by page address registers, one for each 2 Kbyte page of real store. The page address registers are accessible to the system software as V-store locations, and they have the following fields:



a) Lockout

The lockout bit has the effect of invalidating the address field, as far as equivalence on store accesses is concerned. It is used by the system software on two occasions:

i) When the page is not in use by any process.

ii) To prevent access to the virtual address when the page is in transfer to/from the backing store.

b) Virtual address

Each PAR contains a virtual address which uniquely identifies the page of store. Thus, in addition to the segment and block fields, the PARs also contain a process field, which is compared with the V-line V.CURRENT before its most significant digit. The exponent has the base 16, and is interpreted as a 7-bit 2s complement integer plus 64 (2s complement with sign bit inverted), i.e.,

0000000 → -64
0000001 → -63
.
.
1

c) Access Permission

Upon finding an equivalence with the virtual address, the address translation hardware checks the requested access against the access permission field in the PAR. This contains 4 bits, as follows:

```
U - 1 => access specified by  RWX bits available in
         both user and executive modes.
    0 => access specified by  RWX bits available in
          executive mode only.
R - read operand permitted
W - write to operand permitted
X - instruction fetches permitted
```

If the required access is permitted, the appropriate line of store is accessed and the usage statistics in the PAR are updated accordingly. If the requested access is not permitted, a virtual store interrupt is forced.

d) Usage

The usage statistics consist of 2 bits, as follows:

REF – set on any successful access to the page
ALT – set on any successful write access to the page.

7.2 PAR NON-EQUIVALENCE

Non-equivalence of the PARs is possible in three situations:

i) The page requested is not currently in store

ii) The page is in store, but the lockout bit is set in the PAR

iii) The page is in store, but the attempted access violates the access permission in the PAR.

Any of these three gives rise to virtual store interrupt, with V.NEQ.SEG and V.NEQ.BLOCK indicating the failing virtual address and V.NEQ.ACCESS giving the type of access which was attempted (4 bits). The operating system uses this information to determine whether the access was a valid one, and if so to obtain the required page.

7.3 ADDITIONAL FEATURES OF THE MEMORY MANAGEMENT UNIT

The memory management unit provides additional functions, to assist the system software in validating addresses and locating blocks of store with particular characteristics.

i) Address Validation

The VAL order, mentioned in Chapter4, is used for checking the validity of a particular virtual address for the current process. The virtual address is specified as the operand of the order (usually in the form 'base register + displacement'), the permitted access is returned in an internal register VALD in a form equivalent to the URWX permission bits. Apart from this the order is treated as a null operation if the page is in core. Otherwise, a validate fail interrupt occurs, with the failing address indicated in the V-lines V.NEQ.SEG, and V.NEQ.BLOCK as for a PAR non-equivalence.

ii) Context searches

The main functions that the operating system has to perform, in relation to the PARs, are:

a) Locate a particular block by searching for the PAR with the required

(PSB) fields.

b) Release the pages of a segment, by searching for blocks with a given (PS) field, and setting them free.

c) Reduce the size of a segment, by searching for blocks with a given (PS) field and (B) > some value, and setting them free.

d) Change the virtual address, by searching for blocks with a given (PS) field, and changing the (S) field.

e) Change the access permission, by searching for blocks with a given (PS) field, and changing the access permission field.

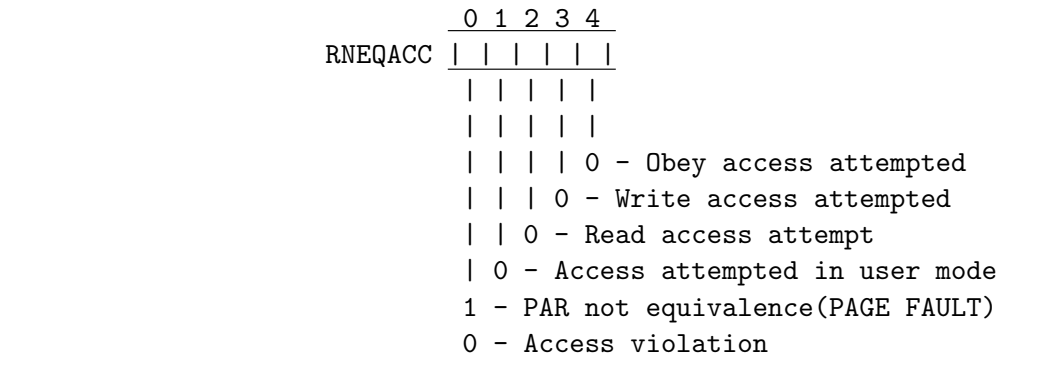
The memory management unit assists in these operations by providing a facility for the context searching of the PARs. The virtual address is set into the V-lines V.SEARCH.PROC, V.SEARCH.SEG, and V.SEARCH.BLOCK, with a fourth V-line V.SEARCH.MASK, controlling the fields to be compared. The search is initiated by writing to V.SEARCH.MASK, and a matching PAR number is returned in V.PAR.NO (a reply > the number of PARs present indicates no match was found). If '0' is written to bit 0, a new search is to be initiated; if '1' is written to bit 0 the search continues from the value in V.PAR.NO.

7.4 SUMMARY OF V-LINES ASSOCIATED WITH MEMORY MANAGEMENT.

NAME	64bit word V-ADDRESS	WRITE/READ	FUNCTION
V.CURRENT.PROCESS.NUMBER	%0100	W/R	(RPRONO)-8 bits. Holds current PROCESS number.
MACOP	%0101	W/R	(RMACOP)-5 bits. Controls the operation of MAC as shown below.

```
                 0 1 2 3 4
RMACOP           | | | | |
                 | | | | |
                 | | | | |
                 | | | | 1 - PAR override [RPAROVR]
                 | | | 1 - CPR override [RCPROVR]
                 | | 1 - CACHE override [RCACOVR]
                 0 0 - NO PREFETCH
                 0 1 - NO PREFETCH
                 1 0 - PARTIAL PREFETCH
                 1 1 - FULL PREFETCH
```

V.NEQ.BLOCK	%0102	W/R	(RNEQBOK)-7 bits. Contains the BLOCK number which caused access violation or PAR not equivalence (page fault interrupt)
V.NEQ.SEG	%0103	W/R	(RNEQSEG)-8 bits. Contains the SEGMENT number which caused access violation or PAR not equivalence
V.NEQ.ACC	%0104	W/R	(RNEQACC)-5 bits. Contains the type of access attempted and reason for failure as shown below.


0 1 2 3 4
RNEQACC
0 - Obey access attempted
0 - Write access attempted
0 - Read access attempt
0 - Access attempted in user mode
1 - PAR not equivalence(PAGE FAULT)
0 - Access violation

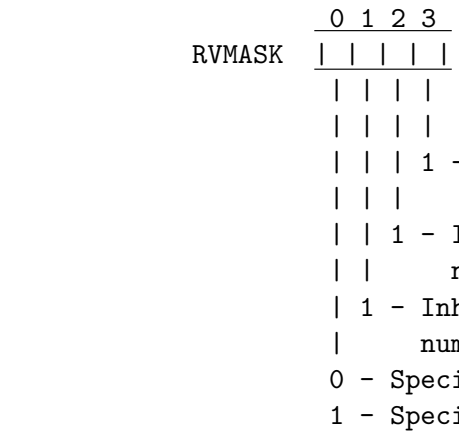
V.SEARCH.PROC
%0105 — (RVPROC)-8 bits. Holds the PROCESS number to be used during a V.SEARCH.

V.SEARCH.SEG
%0106 — (RVSEG)-8 bits. Holds the SEGMENT number to be used during a V.SEARCH.

V.SEARCH.BLOCK
%0107 — (RVBLOCK)-7 bits. Holds the BLOCK number to be used during a V.SEARCH.

V.PAR.NO
%0109 W/R — (RPARNO)-12 bits. Holds a PAR number which is returned as a result of a V.SEARCH or a FREE PAR SEARCH. Can also be loaded before a V.SEARCH or a FREE PAR SEARCH, hence the search can begin from that PAR number.

V.SEARCH.MASK
%0118 W/R — V.SEARCH MASK REGISTER (RVMASK)-4 bits. Holds the V.SEARCH address masking information as shown below and this is used during a V.SEARCH operation.


0 1 2 3
RVMASK
1 - Inhibits use of BLOCK number during a VSEARCH
1 - Inhibits the use of SEGMENT number during a VSEARCH
1 - Inhibits the use of PROCESS number during a VSEARCH
0 - Specifies START VSEARCH
1 - Specifies Continue VSEARCH

Writing to V.SEARCH.MASK initiates the V.SEARCH and depending on bits 01-03, the PROCESS,SEGMENT,BLOCK fields respectively are masked during the search.

During a START search (i.e. setting bit 0 to zero), all the entries in the CACHE and the CPRs which meet the search criterion will be purged.

During a CONTINUE search (i.e. setting bit 0 to one), the V.SEARCH will start from the PAR number held in RPARNO.

V.FREE.PAR

%0119	W/R	Read operation identical to V address 09. Writing V.FREE.PAR initiates a FREE PAR search. This search returns a PAR number in RPARNO of a PAR which is not in use.

V.CPR.PURGE		
%011D	W	No read operation. Writing to this address causes a complete purge of all CPRs.
V.CACHE.PURGE		
%011E	W	No read operation. Writing to this address causes a complete purge of all CACHE entries.
V.PAR.RESET		
%011F	W	No read operation. Writing to this address causes all the PARs USE bits to be reset and the LOCKOUT bits set.
PARs		
%1000-%11FF	W/R	PARs 0-511 covering 1 Mbyte of store. Format as shown at start of Chapter7.
%1000-%13FF	W/R	PARs 0-1023 covering 2 Mbytes of store.
%1000-%1FFF	W/R	PARs 0-8191 covering 8 Mbytes of store.
V.STORE.STATUS		
%F001	R	2 bit local store V-line to inform the operating system of the availability of the stacks. Bit 63 - stack 0 off Bit 62 - only 1 stack available 62,63 00 - normal interleaving 01 - illegal 10 - 0 → n stack 0; (n+1) → 2n stack 1 11 - 0 → n stack 1; (n+1) → 2n stack 0

8 PERIPHERAL & MULTIPROCESSOR CONTROL

PERIPHERALS

The initial implementation of MU6G has a restricted peripheral handling capability. Accordingly, many I/O operations are performed by an additional front-end processor (PDP11/34) connected to the MU6G highway. The peripheral interface units allow limited number of Unibus compatible devices to be connected directly to MU6G. In the initial implementation these include System Industries 300GB disc and Genisco graphical display.

8.1 COMMUNICATION WITH PDP11 – COMPUTER INTERFACE UNIT

The unit is designed to transfer messages of up to 8 Kbytes in length between the stores of the two machines using direct memory accesses, interrupting the machines only at the end of a transfer. Transfers are executed in words of 64 bits, the unit making four accesses to the PDP11 store under a single Non Processor Request for each access to the MU6G store.

Each machine has separate transmit and receive ports containing store addresses and control information. The transmitting machine sets the transfer size, also start/end of message flags for multi-block transfers. Flags are also included which indicate that the other machine has been reset/re-booted.

An engineering mode is available enabling either machine to access all transfer control V-lines and also allowing MU6G to access directly any device on the PDP11 Unibus. 'Eng' beside an address below indicates that the address is only available in engineering mode. Engineering mode must be reset before a block transfer is started.

Block Transfer Unit A transfers from MU6G to PDP11 Unit B from PDP11 to MU6G.

Summary of V-lines:

All lines are read/write but individual bits may be read only or write only. System bus position 5, Vstore unit %B. PDP11 Control words are *7650dd and Interrupt vector is *300.

NAME	MU6G 64 bit word V-address	PDP11 Unibus Byte Address	FUNCTION
	%B000	Eng *765020	MU6G command status

bits 48-63 in MU6G, (bits 15-0 in PDP11)

Bit 48 (15) 'A' Transfer complete - a cause of interrupts to MU6G.
R/W = 1 to reset.

49 (14) PDP11 Hardware reset flag - a cause of interrupts to MU6G.
R/W = 1 to reset.

50 (13) 'A' Start from 11 – PDP11 ready to receive data.
R only.

51 (12) 'A' Start from G – MU6G ready to transmit data.
R/W = 1 to set.

52 (11) 'A' End of message flag to PDP11.
R/W

53 (10) 'A' Start of message flag to PDP11.
R/W

54 (9)	Inhibit interrupts into to MU6G. R/W
55 (8)	MU6G Engineering mode. R/W
56 (7)	'B' transfer complete - a cause of interrupts to MU6G. R/W = 1 to reset.
57 (6)	MU6G Hardware reset flag to PDP11. R only.
58 (5)	'B' start from G – MU6G ready to receive data. R/W = 1 to set.
59 (4)	'B' start from 11 – PDP ready to transmit data. R only.
60 (3)	'B' End of message flag to MU6G. R only.
61 (2)	'B' start of message flag to MU6G. R only.
62 (1)	'B' Inhibit interrupts into PDP11. R only.
63 (0)	PDP has set engineering mode (– go away MU6G). R only.

%B002 Eng *764024 — G.Count 12 bits (52-63) loaded with one less than the number of 16 bit words to be transferred from MU6G to the PDP11 by 'A' unit. Counts down to -1 at the end of transfer.
R/W

	%B004	Eng *764030	16 bits (48-63) – Start address of data within the MU6G store for transmission to PDP11 by 'A' unit. Bit 63 = 128 byte units. This address must be $n2^x$ where $2^x >$ G Count (%B002). R/W
	%B005	Eng *764032	16 bits (48-63) – Start address of the buffer area within the MU6G store for receipt of B transfer data from the PDP11; Bit 63 – 128 Byte units. This address must be $n2^x$ where $2^x >$ P Count (%B00A). R/W
Eng	%B008	*764000	PDP11 Command Status bits 15-0 in PDP11, (Bits 48-63 in MU6)

Bit 15(48) 'B' Transfer complete – a cause of interrupts to PDP11
R/W = 1 to reset

14(49) MU6G Hardware reset flag – a cause of interrupts to PDP11.
R/W = 1 to reset.

13(50) 'B' Start from G – MU6G ready to receive data.
R only.

12(51) B Start from 11 – PDP11 ready to transmit data.
R/W = 1 to set.

11(52) 'B' End of message flag to MU6G.
R/W

10(53) 'B' Start of message flag to MU6G.
`R/W`

9(54) Inhibit interrupts into PDP11.
`R/W`

8(55) PDP11 Engineering mode.
`R/W`

7(56) 'A' Transfer complete – cause of interrupts to PDP11.

6(57) PDP Hardware reset flag to MU6G.
R only.

5(58) 'A' Start from 11 – PDP11 ready to receive data.
`R/W` =1 to set.

4(59) 'A' start from G – MU6G ready to transmit data.
R only.

3(60) 'A' End of message flag to PDP11.
R only.

2(61) 'A' Start of message flag to PDP11.
R only.

1(62) Inhibit interrupts into MU6G.

0(63) MU6G has set engineering mode (go away PDP11).
R only.

Eng	%B00A	*764004	PDP11 12 bits (11-0) loaded with one less than the number of 16 bit words to be sent from the PDP11 to MU6G by the 'B' unit. Counts down to -1 at the end of transfer. R/W
Eng	%B00C	*764010	Base address in PDP11 - the Start address of the data within the PDP11 store for transmission to MU6G by the B unit. 12 bits long (11-0) bit 0 − 64 byte units. This address must be 2^x where $2^x >$ PDP11 Count (%B00A). R/W
Eng	%B00D	*764012	'A' Base address in PDP11 - the Start address of the buffer area within the PDP11 store to receive 'A' transfer data from MU6G. 12 bits long (11-0) bit 0 = 64 byte units. This address must be $n2^x$ where $2^x >$ G Count (%B002). R/W

8.2 PERIPHERAL INTERFACE UNITS

Devices connected include System Industries 300MB disc and Genisco graphical display.

The Peripheral Interface Unit simulates a PDP11 processor and store to the Unibus device attached to it. It contains two address extension registers, one to extend the 12 bit V-store address from MU6G up to the 18 Unibus bits required by the peripheral and one to extend the memory address from the peripheral up to the 2^{20}x64 bit words of MU6G store addressing. There are buffer registers to assemble and dissemble the 16 bit words of data of the Unibus to and from the 64 bit words of MU6G and also a register to indicate the number of grouped direct memory accesses on the Unibus for each MU6G store access. Another register holds the Interrupt Vector issued by the

peripheral until MU6G has used it and no further interrupts are accepted from the peripheral until the register has been freed by MU6G.

8.2.1 System Industries Disc P.I.U.

Summary of V-lines. System bus position 3. Vstore unit %9.

NAME	MU6G 64 bit word V-address	FUNCTION
	%9000	MU6G address extension. 8 bits (56-63) bit 63 – 32Kbyte units. The 3 least significant bits overlap with the 3 most significant of the 18 bit Unibus address (byte). R/W
	%9001	Unibus address extension. 6 bits (58-63), bit 63 - 4Kbyte units. Concatenated with the 12 bit V-store line address to address notionally 64 bit control words within the peripheral. Only the 16 least significant bits of these 64 are used. R/W
	%9002	Interrupt vector. 8 bits (56-63) containing the byte oriented interrupt address issued by the disc. R/W = 1 to clear interrupt.
	%9003	DMA count 4 bits (60-63) set to 2^{n-1} where n is the number of 16 bit transfers / MU6G accesses. Normally set to 8, i.e. 2^3 for 4 16 bit transfers. R/W

For detail of the following lines see System Industries documentation. Unibus address extension (%9001) must be set to %3F.

%9EE0	Control word. 16 bits (48-63).
%9EE1	Word count. 16 bits (48-63). Number of 16 bit words to be transferred. `R/W`
%9EE2	Port/cylinder address. `R/W`
%9EE3	Head/sector address. `R/W`
%9EE4	Memory address (bytes) 16 bits (48-63). `W` only.
%9EE5	Error register. `R` only
%9EE6	Seek Status Register (bytes) 16 bits. (48-63). `R/W`
%9EE7	Seek Address register. `R` only.
%9EE8	Data buffer. `R/W`
%9EE9	Communication Register. `R/W`
%9EEA	Shared Computer Register. `R/W`

8.2.2 Genisco Graphical Display P.I.U.

9 MU6G ASSEMBLY LANGUAGE

The simple assembly language (GAL) described here is fairly compatible with the assembly languages used on other machines in the MU6 network. Its main characteristics are as follows:

1) Each instruction in assembly language corresponds exactly to one machine instruction.

2) The function parts of instructions are expressed by their mnemonics (e.g. `ADD, INE`, etc.).

3) Store lines whose addresses appear explicitly in instructions are given names.

4) Values can be associated with names by one of two different mechanisms. First a name can be attached as a label to an instruction and its value is then the address of the store line containing the instruction. Second a value, which is a function of the values of other names and constants, can be assigned directly to a name by means of a statement called a 'label setting directive'.

5) Constants can appear in a number of contexts − for example, as an operand, as the address of an operand or in a list of constants embedded in the program. The constants arise naturally in a variety of forms − for example, decimal, hexadecimal, octal or the codes for characters and short character strings. The assembly language provides a notation for each of these variations.

6) A block structure is provided to localise the scope of a name. Outside the block in which a name is defined, it cannot be referenced unless it is defined again. Usually each sub-routine is expressed as a separate block.

9.1 A GAL PROGRAM

A program in GAL is a sequence of statements terminated by an enter statement. It usually consists of several subroutines delimited by '`BEGIN`' and '`END`' and a main program sequence. In complicated programs, the block

structure might extend through several levels of nested ‘BEGIN’s and ‘END’s. It might be helpful to illustrate the language with an example of a GAL program before giving the detail rules of the language. The example given is a program which, by working in BCD, reads pairs of positive, and not greater than 8 digit, decimal numbers, adds them and prints a decimal answer. It stops when a zero is read and assumes library routines for INCH, OUTCH and STOP.

```
BCDADD:BEGIN; BM2=BM2+(BCD) BM3
      LDM1 BM2
      NEQ1 BM3  ; BM1=BM2 /= BM3
      ADD2 BM3
      ADD2 LIT1;  BM2=BM2+BM3+%66666666
      NEQ1 BM2
      AND1 LIT1+1
      SFT1 #2
      SUB2 BM1  ; subtract excess 4's
      SFT1 #1
      SUB2 BM1  ; subtract excess 2's
      RTS  @-1/4
 LIT1: # %66666666
       # %11111111
      END
BCDREAD:BEGIN; BM2 = UNSIGNED BCD NUMBER
   L1:LDM2 #0
      JSR  #INCH
      CMP1 #'0
      IMI →L1
      CMP1 #'9
      IHI →L1 ; IGNORE CHS UNTIL DD
   L3:AND1 #15
      SFT2 #-4
      ADD2 BM1 ; BUILD UP BCD NO
      JSR  #INCH
      CMP1 #'0
      IMI →L2
      CMP1 #'9
      ILO →L3 ; REPEAT UNTIL NO MORE DD'S
   L2:RTS  @-1/4
      END
```

```
BCDPRINT:BEGIN; PRINT 'NL' THEN NO IN BM2
              ; AS 8 BCD DIGITS
      LDM1 #10
      JSR  #OUTCH ; PRINT NL
      LDM3 #1 ; SET COUNT
   L1:SFT2 #CIR-4
      LDM1 BM2
      AND1 #15
      ADD1 #'0 ; COMPUTE NEXT CH IN BM1
      JSR  #OUTCH
      CIN3 #8
      INE →L1
      RTS  @-1/4
      END
 START:RJSR →BCDREAD
      LDM3 BM2
      IEQ →L1
      RJSR →BCDREAD
      RJSR →BCDADD
      RJSR →BCDPRINT
      BR →START
   L1:JSR  #STOP
      ENTER START
```

Assembly language is meant to be a tolerably readable form of machine instructions and it is hoped that the above example demonstrates this. In detail the statements in a GAL program may be

```
Instructions
Constant Statements
or Directive Statements.
```

These three types of statement will be described separately, but constants which are common to all are described first.

9.2 CONSTANTS

A constant can be expressed as any of the following forms of basic constant, or as an expression involving several basic constants

```
decimal constant     (e.g.  10, 1023)
hexadecimal constant (e.g.  %A, %FFFF
octal constant       (e.g.  #12, #1777)
character constant   (e.g.  'A, '7)
label                (e.g.  L1, RDCH)
```

Decimal constants are considered to be the norm, therefore any sequence of decimal digits not preceded by a qualifying symbol will be interpreted as a decimal constant. The onus is on the programmer to only use constants that will be in range when converted to binary. Constants which are to appear in instructions must not exceed 16 bits, in tables they may be 32 bits.

Hexadecimal constants are preceded by the qualifying character % after which any sequence of hexadecimal digits is allowed. They will be right justified. Thus

```
%1          is equivalent to   %00000001
%3FF        is equivalent to   %000003FF
%555555555  is equivalent to   %55555555
```

Octal constants are preceded by the qualifying character * after which any sequence of octal digits is permitted. The result generated is again right justified. That is to say

```
*1          is equivalent to *00000000001
*1777       is equivalent    *00000001777
```

Character constants are provided to avoid the need to refer to the table in Appendix 2[1] whenever the code corresponding to a character of input or output is required. The qualifying character is ' after which any character may be given. The constant generated by the assembler will be the value of the given character in the table in Appendix 2. Thus

```
'A is equivalent to *101 or %41
'7 is equivalent to *67 or %37.
```

Labels are names which have been given values elsewhere in the program. When used as constants their values are substituted. Names always begin

[1] ed.: This evidently refers to a standard ANSI code table such as the Table in Appendix 1 of the MUSS USER MANUAL Vol 1.

with a letter which may be followed by up to seven further letters and/or digits. If a constant involves a name which has been earlier defined, this is called a backward reference, whereas if the name is later defined, it is called a forward reference. There are some contexts in which a forward reference is not allowed and these will be identified as they arise.

Expressions are sequences of basic constants connected by the operators

```
+    (meaning add)
-    (meaning subtract)
%    (meaning logical 'and')
!    (meaning logical (inclusive) 'or')
→    (meaning right logical shift)
←    (meaning left logical shift)
```

and possibly preceded by a minus sign.

The assembler evaluates each expression as it reads it to generate the equivalent 32-bit constant. Depending on the context in which it appears it may then be truncated but this is described later, and the values of labels appearing in expressions may sometimes be scaled, but this is also explained later. A legitimate but frivolous example is

```
-1&*177400!'A+256.
```

This would be evaluated 'left to right' with all operators having equal precedence to yield

```
(((-1)&*177400)!'A)+256 = *101.
```

More realistic examples usually involve labels and are often quite simple, for example

```
L1+2
COUNT-1.
```

It should be noted that forward references are not permitted in expressions.

9.3 INSTRUCTIONS

The main instruction form in GAL is

```
MNEMONIC     OPERAND
```

where the mnemonic may be chosen from the table

ST	STM1	STM2	STM3	JSR	RJSR	XJSR	RTS
STX	spare	spare	spare	VAL	STK	spare	STR
LDR	LDM1	LDM2	LDM3	LMS	spare	JMP	spare
LDD	NEQ1	NEQ2	NEQ3	spare	STA3	LDA3	LRA3
SP1	AND1	AND2	AND3	spare	STA4	LDA4	LRA4
spare	OR1	OR2	OR3	spare	STA5	LDA5	LRA5
LDX	OPR1	OPR2	OPR3	spare	STA6	LDA6	LRA6
LSR	CIN1	CIN2	CIN3	BR	STA7	LDA7	LRA7
ADD	ADD1	ADD2	ADD3	IEQ	STA8	LDA8	LRA8
SUB	SUB1	SUB2	SUB3	INE	STA9	LDA9	LRA9
RSB	RSB1	RSB2	RSB3	IPL	ST10	LD10	LR10
MLT	MLT1	MLT2	MLT3	IMI	ST11	LD11	LR11
DIV	DIV1	DIV2	DIV3	ILO	ST12	LD12	LR12
RDV	RDV1	RDV2	RDV3	IHI	ST13	LD13	LR13
SCL	SFT1	SFT2	SFT3	IBV	ST14	LD14	LR14
CMP	CMP1	CMP2	CMP3	IBC	ST15	LD15	LR15

An operand can have one of several forms that correspond to the operand types provided for by the 'K' bits in the hardware.

9.4 OPERANDS

There are five operand forms which correspond with the five different forms of 'K', and a special form for use in the branch instruction where it is necessary to generate a 16-bit word relative address. The first four forms always involve a literal which may be optionally BA or BM modified. Although a zero modifier can always be specified to indicate that no modification is required, the language allows either BM or both BA and BM to be omitted.

Thus using <C> to indicate any constant the allowed forms for the literal /BA/BM parts are

```
<C>          e.g. 99    meaning literal 99  BA = 0 BM = 0
<C>/<C>      e.g. 100/1 meaning literal 100 BA = 1 BM = 0
<C>/<C>/<C> e.g. 110/9/2 meaning literal 110 BA = 9 BM = 2
                 120/0/3 meaning literal 120 BA = 0 BM = 3
```

Denoting the above forms of modified constant as <MC> the six permitted operand forms are

```
#<MC>    meaning k = 0 i.e.  literal
$<MC>    meaning k = 1 i.e.  string operand
<MC>     meaning k = 2 i.e.  word operand
@<MC>    meaning k = 3 i.e.  double word operand
<REGISTER>
→<C>     meaning treat as branch address.
```

The fifth alternative corresponds the register form of operand. For example

```
LDA7  BA8
```

sets BA7 equal to BA8. A full list of register names is given in Ch.2.4. These names are reserved and must not be used as labels.

Some further explanation of the treatment of labels in constants is now appropriate. Addresses in MU6G refer to bits whereas modifiers and literal displacements are in units of the size of object being addressed. That is to load a 32-bit word whose address is given by BA7 into BM1 we use the instruction

```
LDM1 0/7.
```

If the word in question is word 10 then BA7 must contain %140. However to access the next word we would write

```
LDM1 1/7.
```

Similarly if BM2 was known to contain 10 then word 10 might be loaded into BM1 thus

```
LDM1 0/0/2.
```

If we now consider the case where the address of the required word is given by a label (L1) we might expect to be able to access the word

```
by LDM1 L1
or LDA7 #L1
   LDM1 0/7.
```

This will only be possible if the assembler automatically scales L1 in one case or the other. The decision taken has been to always associate with labels addresses in bit units, and scale as follows

```
in #<MC>      no scaling
in $<MC>      scale by 2^3
in <MC>       scale by 2^5
in @<MC>      scale by 2^6
in →<C>       scale by 2^4.
```

However, this scaling applies only to labels which are associated with addresses. For example, given that L1 is the address of word 10 (or %140)

```
LDMI   L1+1
```

would also access word eleven. Since literals are not scaled the programmer may sometimes need to scale explicitly. For example, two alternative ways of accessing the word L1 are

```
LDA7  #L1 or LDM2 #L1→5
LDM1  0/7    LDM1 0/0/2.
```

It remains to define what is meant by 'a label associated with an address', and this is is done in the section on directive statements.

If L1 is in the code segment, and this is a segment other than zero the access is automatically converted by the assembler to be /C0, as otherwise L1 could not be represented in 16 bits.

9.5 CONSTANT STATEMENTS

When it is required to include a list of constants within a program, each constant is written as a separate statement preceded by the hash sign. For

example

```
POWERS: #1
        #10
        #100
        #1000
        #10000
```

would cause a table of powers of 10 to be assembled into a program (each constant occupying 32 bits) and assign the address of the first (10^0) to the value of the label `POWERS`. It may also be required to store a string of characters within a program and this can be achieved by enclosing the string in quotes e.g.

```
# "ADCDEF".
```

Constants are planted on the next available 32 bit boundary in the code.

9.6 DIRECTIVE STATEMENTS

These statements direct the assembly process without causing any instruction to be assembled into the program. `BEGIN` and `END` are clearly in this category. As has already been stated, they delimit the block of statements that they contain. The significance of this is that the definitions of all the labels defined within the block are erased at the end of the block. The most common way of assigning a value to a label is by 'labelling' a statement. A label on a statement simply has the form

```
name:
```

and this can be thought of as a directive statement. `ENTER` is another directive and it marks the end of the program. It must be followed by a constant which gives the entry address for the program.

One directive remains, it is the 'label setting directive'. The general form of this statement is

```
name = constant.
```

Examples are

```
COUNT = 10
L3    = L4
L7    = L10+8.
```

There are some peculiarities of this type of statement which should be noted. First, it is the only statement that violates the normal block structure rule relating to the scope of names. Normally the use of a name in a constant refers to a definition of the name (either before or after this use) in the same block, or failing that in the first enclosing block and so on. The exceptional feature of the label setting directive is that the names in the constant part cannot be forward references. For each such name the nearest (in a block structure sense) prior definition is always used. Perhaps this can be made clear by an example

```
COUNT  = 10
COUNT2 = 20
BEGIN
COUNT1 = COUNT
COUNT  = 30
LDM1 #COUNT
LDM2 #COUNT1
LDM3 #COUNT2.
```

This will result in instructions which set

```
BM1 = 30
BM2 = 10
BM3 = ?
```

The label setting directives always generate an explicit value for the label, hence, inside the block `COUNT1` and `COUNT` have values 10 and 30. All references to names such as these which have been defined in the current block are replaced by their values. References to names not yet defined in the current block (e.g. `COUNT2`) are treated as forward references, and are left blank at least until the end of the block, unless they are defined within the block. At the end of the block `COUNT 1` and `COUNT` cease to be defined as 10 and 30, `COUNT1` becomes undefined and `COUNT` reverts to its previous value of 10. Also the definition of `COUNT2` becomes operative again and the blank literal previously left in the program becomes 20.

Forward references are treated as 16 bit literals, except in constant statements where 32 bits are allowed. Note that all forward branches to labels are therefore 32 bit instructions. `BR+1` would however result in a correct 16 bit instruction.

In an earlier section the notion of 'a label associated with an address' was used but not defined. Any label set by attachment to a statement (e.g. name:) is clearly associated with an address. Other labels set by the label setting directive are only associated with an address if the expression on the r.h.s. involves any labels which themselves are associated with addresses.

9.7 SPECIAL NAME 'DOT'

The symbol '.' is treated as a pseudo name and the value associated with it is always the address of the store line into which the next statement will be assembled. Although it can be used in any context where a normal name is permitted the most common use is in controlling the position of code and data at assembly time. For example the statement

```
.  = *2000 ← 5
```

will result in the assembly of subsequent statements into store line 2000 (octal) onwards. Another fairly common use is in leaving areas of store empty to be used as working space. For example, two areas each of size 100 32-bit store lines called `AREA1` and `AREA2` can be created using

```
AREA 1:
.  = 100 ← 5 + .
AREA2:
.  = 100 ← 5 + .
```

The GAL assembler will only assemble code into a single MU6G segment and DOT should only be moved within that segment.

9.8 COMMENTS

A comment can be placed on any line of a program by using the character ';'. After a semicolon all characters up to the next newline will be ignored, except in string constants.

9.9 GAL ASSEMBLER

The GAL procedure is called with 3 parameters

P1 = input source file name
P2 = output binary file name
P3 = run time code segment address.

Bibliography

[1] Edwards, D. B. G., and Knowles, A. E., and Woods, J. V. MU6G: A new design to obtain mainframe performance from a mini-sized computer. *ACM 7th Annual Symposium on Computer Architecture* (May 1980).

[2] Theaker, C. J. and Brookes, G. R. *A Practical Course on Operating Systems*. MacMillan Computer Science Series. MacMillan, London, UK, 1983. pgs. 1–203.

Index

., 101

A-UNIT, 25
Accumulator Orders, 8
ADD, 8, 25
ADD1, 9, 23
AEX, 25
AEXP, 25
Algol, 14, 30, 45, 47–51
ALGOL68, 2
AND1, 9
AOD, 27
ATLAS, 1, 2, 58

BA, 2, 3, 11, 12
BA15, 67
BA3, 10
BA4, 10
BA4–15, 10
BC, 18, 19
BCD, 9
BITA, 15
BM, 2, 11, 16, 19, 20
BM Orders, 9
BM UNIT, 19
BM1, 9, 11
BM2, 11
BM3, 11
BR, 10
BREM, 19, 24
BTOP, 19
BV, 18

CDC7600, 1
CIN1, 9, 23
CMP, 8
CMP1, 9, 23
CO, 12, 16
COBOL, 2, 60
Control Orders, 10

DEC, iii
DIV, 8, 9
DIV1, 24
DOT, 101

EQ, 18

FORTRAN, 2, 3, 5, 14, 30, 43, 48, 49, 54

GAL, 91, 101

HI, 18

IBC, 10
IBM, 25
IBV, 10
ICL, iii, 25
IEQ, 10
IHI, 10
Iliffe vector, 45
ILO, 10
IMI, 10
INE, 10
Interrupts, 71
IPL, 10

JMP, 10
JSR, 10

KA10, 6

LDA3, 10
LDA4, 10
LDA4–15, 10
LDD, 8
LDM1, 9
LDR, 8

LDX, 8
LMS, 10
LO, 18
LRA3, 10
LRA4, 10
LRA4–15, 10
LSR, 8

MI, 18
MLT, 8
MLT1, 9, 24
Mode Bits, 27
Motorola, iii
MS, 18
MU4, 1
MU5, iii, 1, 2
MU6, iii, 1, 4, 91
MU6G, iii, iv, 1, 3, 6, 75, 88, 97, 101
MUSL, 2
MUSS, iii, iv, 2
MUSS USER MANUAL, 94

NB, 12, 66
NE, 18
NEQ1, 9

Operand Classes
  from literals, 13
  from registers, 13, 14
  from store, 13
Operand Forms, 10
Operands, 13
  Variable, 15
  Literal, 17
  Register, 17
OPR1, 9
OR1, 9
Order code, 2
Orders, 10
  Accumulator, 8
  BM, 9
  Computational, 3
  Control, 10
  Organisational, 3
  Other Organisational, 10

PAGE FAULT, 80
PAR, 75–81
PDP11/34, 1
Performance, 3
PL, 18

R, 27
RDIV1, 24
RDV, 8, 9
RJSR, 10
RSB, 8
RSB1, 9, 24
RTS, 10, 38, 65

SCL, 8
SF, 12, 66
SFT1, 9, 24
SP1, 8
ST, 8
STA3, 10
STA4–15, 10
STK, 10
STM1, 9
STR, 10
STX, 8
SUB, 8
SUB1, 9, 23

UMIST, 6

V.MC.CONTROL, 65
V.PRB.A, 66
V32, 66
VAL, 10
VM.TIMER, 66

XJSR, 65